COP509

Natural Language Processing

Text Cleaning and Preprocessing

Prof. Georgina Cosma

Department of Computer Science
Loughborough University

g.cosma@lboro.ac.uk

January 2026

Learning Outcomes

By the end of this session, you will be able to:

1. Explain why text cleaning is essential before any NLP task
2. Apply manual text cleaning techniques using Python
3. Use NLTK for tokenisation and text preprocessing
4. Remove punctuation, normalise case, and filter stop words
5. Apply stemming to reduce words to their root forms
6. Understand what word embeddings are and how cleaning affects them
7. Choose appropriate cleaning strategies for different NLP tasks

Outline

1. Why Clean Text?
2. Understanding Your Text Data
3. Manual Text Cleaning with Python
4. Tokenisation and Cleaning with NLTK
5. Stop Words and Stemming
6. Putting It All Together: A Cleaning Pipeline
7. Understanding Word Embeddings
8. Cleaning for Word Embeddings
9. Summary

Part 1

Why Clean Text?

The Problem: Computers Don't Understand Text

Remember: Computers only understand numbers, not words!

Before we can use text for machine learning, we need to:

1. **Clean** the text – remove noise and irrelevant content
2. **Tokenise** – break text into smaller units (words or subwords)
3. **Normalise** – make text consistent (e.g., lowercase)
4. **Convert** – transform words into numbers

Why This Matters

Garbage in = Garbage out! If your text is messy, your model will learn from noise instead of meaningful patterns.

What is “Dirty” Text?

Real-world text is messy! Here are common problems:

Formatting Issues:

- Mixed case (“Hello” vs “HELLO” vs “hello”)
- Extra whitespace and line breaks
- Special characters
- HTML tags and markup

Punctuation Problems:

- Commas, periods attached to words
- Quotes and apostrophes
- Hyphens (“armour-like”)

Language Variations:

- Contractions (“don’t”, “it’s”)
- Abbreviations (“Mr.”, “Dr.”)
- Numbers and dates
- Misspellings and typos

Noise:

- Stop words (“the”, “a”, “is”)
- Headers, footers, copyright text
- Irrelevant metadata

Text Cleaning is Task-Specific

Important: There is no one-size-fits-all cleaning approach!

Your cleaning strategy depends on your **task**:

Task	Cleaning Strategy
Language modelling	Keep case, punctuation, style
Document classification	Remove case, punctuation, use stemming
Sentiment analysis	Keep emoticons, exclamation marks!
Named entity recognition	Keep capitalisation (“Apple” vs “apple”)
Word embeddings	Minimal cleaning, keep contractions

Key Principle: Use your task as the lens by which to choose how to clean your text data.

Part 2

Understanding Your Text Data

Always Examine Your Data First!

Before cleaning, always look at your text! Ask yourself:

1. **What language is it?** (English, mixed languages?)
2. **What format?** (Plain text, HTML, PDF, JSON?)
3. **Are there special characters?** (Unicode, emojis?)
4. **What patterns exist?** (Headers, sections, numbering?)
5. **Are there errors?** (Typos, OCR mistakes?)

Why This Matters

- Helps you design an appropriate cleaning strategy
- Prevents accidentally removing important information
- Saves time by targeting actual problems, not imagined ones

Example: Examining "Metamorphosis" by Kafka

Looking at our example text, we observe:

Good news:

- Plain text (no markup!)
- No obvious typos
- Proper English text

Challenges:

- Lines wrapped at ∼70 characters
- Punctuation attached to words
- Contractions ("What's")
- Hyphenated words ("armour-like")

Observations:

- UK English spelling ("travelling")
- Names present ("Mr. Samsa")
- Em dashes used for pauses
- Section markers ("II", "III")
- Quotes and apostrophes

This analysis helps us decide: What to keep? What to remove? What to transform?

Part 3

Manual Text Cleaning with Python

Step 1: Loading Text Data

First, we need to load our text file into Python:

```
# Open the file in read mode ('r') with text encoding ('t')
file = open("metamorphosis_clean.txt", 'rt')

# Read the entire contents into a string
text = file.read()

# Always close the file when done!
file.close()
```

What This Does

- `'rt'` = read text mode (as opposed to binary)
- `file.read()` loads the **entire file** into memory as one string
- For large files, you may need to read line-by-line instead

Step 2: Split by Whitespace

Goal: Convert the text string into a list of words (tokens).

The simplest approach – split on whitespace:

```
# Split into words by whitespace (spaces, tabs, newlines)
words = text.split()
print(words[:10])
```

Output:

```
['One', 'morning,', 'when', 'Gregor', 'Samsa', 'woke',
 'from', 'troubled', 'dreams,', 'he']
```

Pros:

- Simple and fast
- Keeps contractions ("wasn't")
- Keeps hyphenated words

Cons:

- Punctuation attached ("morning,")
- "dreams," $\neq$ "dreams"

Step 3: Split Using Regular Expressions

Alternative: Use regex to split on non-word characters.

```
import re

# \W+ matches anything that is NOT a letter, digit, or underscore
words = re.split(r'\W+', text)
print(words[:10])
```

Output:

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke',
 'from', 'troubled', 'dreams', 'he']
```

Pros:

- Removes punctuation
- Clean word boundaries

Cons:

- "armour-like" → "armour", "like"
- "What's" → "What", "s"

Step 4: Removing Punctuation

Better approach: Split by whitespace, then remove punctuation from each word.

```
import string

# See what punctuation characters exist
print(string.punctuation)
# Output: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

# Create a translation table that removes punctuation
table = str.maketrans('', '', string.punctuation)

# Split into words, then remove punctuation from each
words = text.split()
stripped = [word.translate(table) for word in words]
print(stripped[:10])
```

Output:

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', ...]
```

Result: "What's" → "Whats" (keeps contractions together!)

Step 5: Normalising Case

Why lowercase? Reduces vocabulary size and treats words consistently.

```
# Convert all words to lowercase
words = text.split()
words = [word.lower() for word in words]
print(words[:10])
```

Output:

```
['one', 'morning,', 'when', 'gregor', 'samsa', 'woke',
 'from', 'troubled', 'dreams,', 'he']
```

Trade-off Warning!

Lowercasing loses some information:

- "Apple" (company) vs "apple" (fruit)
- "US" (country) vs "us" (pronoun)
- Sentence boundaries (capital at start)

Consider your task before deciding!

Part 4

Tokenisation and Cleaning with NLTK

What is NLTK?

NLTK = Natural Language Toolkit

A comprehensive Python library for NLP that provides:

- Tokenisation (splitting text into words/sentences)
- Stop word lists for multiple languages
- Stemming and lemmatisation tools
- Part-of-speech tagging
- Named entity recognition
- And much more!

Why Use NLTK?

- **Handles edge cases** that simple string splitting misses
- **Language-aware** – knows about English grammar
- **Well-tested** – used by researchers worldwide
- **Extensible** – easy to customise

Installing and Setting Up NLTK

Step 1: Install NLTK (if not already installed)

```
pip install nltk
```

Step 2: Download the required data packages

```
import nltk

# Download the 'popular' collection (recommended for beginners)
nltk.download('popular')

# Or download specific packages:
nltk.download('punkt')      # For tokenisation
nltk.download('stopwords')  # For stop word lists
```

Note

You only need to download these packages **once**. They are saved to your computer for future use.

Sentence Tokenisation with NLTK

Goal: Split text into individual sentences.

```
from nltk import sent_tokenize

# Load and split into sentences
sentences = sent_tokenize(text)
print(sentences[0])
```

Output:

```
One morning, when Gregor Samsa woke from troubled dreams,
he found himself transformed in his bed into a horrible vermin.
```

Why sentence tokenisation? Some models (like Word2Vec) work better with sentence-level input.

Word Tokenisation with NLTK

Goal: Split text into individual words/tokens.

```
from nltk.tokenize import word_tokenize

tokens = word_tokenize(text)
print(tokens[:15])
```

Output:

```
['One', 'morning', ',', 'when', 'Gregor', 'Samsa', 'woke',
 'from', 'troubled', 'dreams', ',', 'he', 'found',
 'himself', 'transformed']
```

Notice: NLTK's word_tokenize:

- Separates punctuation as individual tokens (“,” is its own token)
- Handles contractions smarter (“don’t” $\rightarrow$ “do” + “n’t”)
- Handles edge cases better than simple splitting

Part 5

Stop Words and Stemming

What Are Stop Words?

Stop words are extremely common words that usually do not contribute to the meaning of a document.

Examples: "the", "a", "an", "is", "are", "was", "were", "to", "of", "in", "for", "on", "with"

Why remove them?

- Reduces vocabulary size
- Speeds up processing
- Focuses on "content" words
- Improves some models

When to keep them?

- Sentiment ("not good" vs "good")
- Phrase detection
- Language modelling
- Some neural models

Think Before Removing!

"To be or not to be" without stop words becomes: "be be" – completely meaningless!

Removing Stop Words with NLTK

```
from nltk.corpus import stopwords

# Load the English stop words list
stop_words = set(stopwords.words('english'))

# See some examples
print(list(stop_words)[:10])
```

Output (partial):

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'you', ...]
```

Filter out stop words:

```
# Keep only words that are NOT in the stop words list
words = [w for w in tokens if w.lower() not in stop_words]
```

Note: Stop words are lowercase, so convert your tokens to lowercase when comparing!

What is Stemming?

Stemming reduces words to their root or base form (the "stem").

Examples:

Original Words	$\longrightarrow$	**Stem**
fishing, fished, fisher	$\longrightarrow$	fish
troubled, troubling, troubles	$\longrightarrow$	troubl
running, runs, ran	$\longrightarrow$	run
connection, connected, connecting	$\longrightarrow$	connect

Why stem?

- Reduces vocabulary size dramatically
- Groups related words together
- Useful for document classification, search engines

Note: Stems are not always real words ("troubl" is not English!)

Important: Stemming is NOT Reversible!

Common Student Mistake

Many students assume they can stem text for processing, then "unstem" it later to display nice results. **This is not possible!**

Why stemming cannot be reversed:

- "running" $\rightarrow$ "run" – but "run" could have come from: running, runs, ran, runner
- "troubl" $\rightarrow$??? – could be: trouble, troubled, troubling, troubles
- The stemmer does not remember which original word was used

The solution: Keep both versions!

1. Store the **original text** for display to users
2. Create a **stemmed version** for your model/search index
3. Use the stemmed version for matching/classification
4. Display results using the original text

Example: A search engine stems queries and documents for matching, but shows users the original document text in results.

Practical Pattern: Keeping Original and Stemmed Text

How to implement this in practice:

```
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import word_tokenize

# Original text - KEEP THIS for display
original_text = "The cats were running quickly through the garden"

# Tokenize
original_tokens = word_tokenize(original_text)
# ['The', 'cats', 'were', 'running', 'quickly', 'through',
#  'the', 'garden']

# Create stemmed version for processing
porter = PorterStemmer()
stemmed_tokens = [porter.stem(word) for word in original_tokens]
# ['the', 'cat', 'were', 'run', 'quickli', 'through',
#  'the', 'garden']

# Now you have BOTH versions:
# - Use stemmed_tokens for your ML model or search index
# - Use original_text when showing results to users
```

Key takeaway: Never throw away your original text if you need to display it later!

Stemming with NLTK: Porter Stemmer

The **Porter Stemmer** is the most popular stemming algorithm.

```
from nltk.stem.porter import PorterStemmer

# Create a stemmer object
porter = PorterStemmer()

# Stem individual words
print(porter.stem('fishing'))   # Output: fish
print(porter.stem('troubled'))  # Output: troubl
print(porter.stem('running'))   # Output: run

# Stem all tokens in a list
stemmed = [porter.stem(word) for word in tokens]
print(stemmed[:10])
```

Output:

```
['one', 'morn', ',', 'when', 'gregor', 'samsa', 'woke', ...]
```

Stemming vs Lemmatisation

	Stemming	**Lemmatisation**
Method	Chops off word endings using rules	Looks up dictionary form
Speed	Fast	Slower
Output	May not be a real word	Always a real word
Example	"studies" → "studi"	"studies" → "study"
Example	"better" → "better"	"better" → "good"
Use when	Speed matters, rough grouping OK	Accuracy matters

NLTK Lemmatiser

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize('studies')) # Output:  study
```

Part 6

Putting It All Together: A Cleaning Pipeline

A Complete Text Cleaning Pipeline

Standard cleaning steps (in order):

1. **Load** the raw text from file
2. **Tokenise** – split into words using NLTK
3. **Lowercase** – convert all words to lowercase
4. **Remove punctuation** – strip punctuation from tokens
5. **Filter non-alphabetic** – remove numbers and symbols
6. **Remove stop words** – filter out common words
7. **Stem (optional)** – reduce words to roots

Note

Not every step is always needed. Choose based on your task!

Complete Pipeline Code

```
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

# 1. Load the text
with open("metamorphosis_clean.txt", 'rt') as file:
    text = file.read()

# 2. Tokenize
tokens = word_tokenize(text)

# 3. Convert to lowercase
tokens = [w.lower() for w in tokens]

# 4. Remove punctuation
table = str.maketrans('', '', string.punctuation)
stripped = [w.translate(table) for w in tokens]

# 5. Remove non-alphabetic tokens
words = [word for word in stripped if word.isalpha()]

# 6. Filter out stop words
stop_words = set(stopwords.words('english'))
words = [w for w in words if w not in stop_words]

print(words[:20])
```

Pipeline Output

Before cleaning:

```
"One morning, when Gregor Samsa woke from troubled dreams,
he found himself transformed in his bed into a horrible
vermin. He lay on his armour-like back..."
```

After cleaning:

```
['one', 'morning', 'gregor', 'samsa', 'woke', 'troubled',
 'dreams', 'found', 'transformed', 'bed', 'horrible',
 'vermin', 'lay', 'armourlike', 'back', 'lifted', ...]
```

What changed?

- All lowercase
- Punctuation removed ("morning," $\rightarrow$ "morning")
- Stop words removed ("when", "from", "he", "in", "his", "a", "on")
- "armour-like" became "armourlike" (hyphen removed)

Part 7

Understanding Word Embeddings

What Are Word Embeddings?

Word embeddings are a way to represent words as **dense numerical vectors** that capture meaning.

The Problem They Solve:

- Computers need numbers, but simple numbering (cat=1, dog=2, fish=3) does not capture that "cat" and "dog" are more similar than "cat" and "fish"
- We need a representation where **similar words have similar numbers**

The Solution:

- Represent each word as a **vector** (list) of numbers
- Typically 100–300 numbers per word
- Words with similar meanings will have similar vectors

Example:

- "cat" → [0.2, 0.8, 0.1, -0.3, ..., 0.5] (300 numbers)
- "dog" → [0.3, 0.7, 0.2, -0.2, ..., 0.4] (similar to cat!)
- "car" → [-0.5, 0.1, 0.9, 0.4, ..., -0.2] (different from cat)

How Are Word Embeddings Created?

Word embeddings are **learned from large amounts of text** using neural networks.

Key Idea: "You shall know a word by the company it keeps" (J.R. Firth, 1957)

Words that appear in similar **contexts** should have similar meanings:

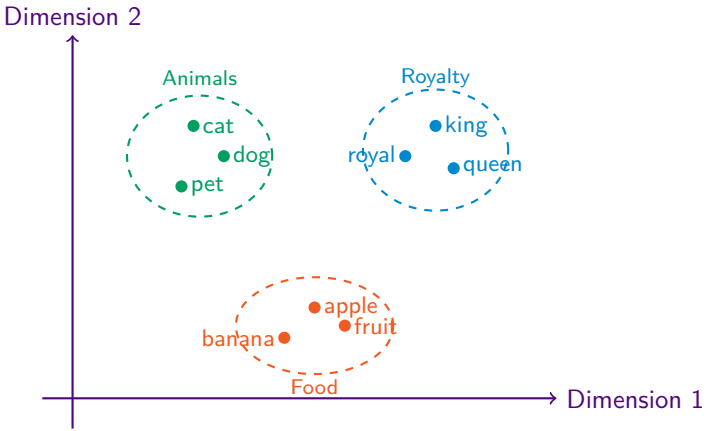
- "The cat sat on the mat"
- "The dog sat on the mat"
- Both "cat" and "dog" appear in similar contexts $\rightarrow$ similar vectors

Training Process (simplified):

1. Take a huge corpus of text (e.g., all of Wikipedia)
2. For each word, look at its surrounding words (context)
3. Train a neural network to predict context from word (or vice versa)
4. The learned weights become the word vectors

Popular algorithms: Word2Vec, GloVe, FastText

Visualising Word Embeddings



Key insight: Similar words cluster together in the embedding space! (Note: Real embeddings have 100–300 dimensions, not just 2)

The Magic of Word Embeddings

Word embeddings capture **semantic relationships** through vector arithmetic!

Famous example:

$$\text{king} - \text{man} + \text{woman} \approx \textbf{queen}$$

More examples:

- Paris − France + Italy ≈ Rome
- walked − walk + swim ≈ swam
- bigger − big + small ≈ smaller

Why This Matters for NLP

- Embeddings capture meaning, not just spelling
- Models can generalise to words not seen during training
- Pre-trained embeddings (trained on huge corpora) can be reused
- Foundation for modern NLP: transformers, BERT, GPT all use embeddings

Part 8

Cleaning for Word Embeddings

Why Embeddings Need Different Cleaning

Key insight: Word embeddings **learn** from patterns in text, so they can handle variations automatically!

What embeddings can learn:

- "running", "runs", "ran" are related → no need to stem!
- "Apple" (company) vs "apple" (fruit) are different contexts
- Context words like "the" and "a" help determine meaning
- Punctuation patterns can signal sentence structure

If you over-clean:

- Stemming destroys information the model could learn
- Removing stop words removes valuable context
- Lowercasing conflates different meanings

Rule of thumb: For embeddings, clean less and let the model learn more!

When TO Stem vs When NOT to Stem

Stemming is useful when:

- Using **bag-of-words** or **TF-IDF** representations
- Building a **search engine** (query "running" should match "runs")
- Doing **document classification** with traditional ML (SVM, Naive Bayes)
- You have **limited training data** and need to reduce vocabulary
- The exact word form does not matter for your task

Stemming is NOT recommended when:

- Training **word embeddings** (Word2Vec, GloVe, FastText)
- Using **neural language models** (BERT, GPT, transformers)
- Doing **text generation** (you need proper words!)
- **Part-of-speech tagging** or **grammatical analysis**
- The word form carries important meaning (e.g., tense matters)

Remember: The "DO NOT STEM" advice applies specifically to word embeddings and neural models, not to all NLP tasks!

Minimal Cleaning for Embeddings

For word embeddings, often less cleaning is better:

Cleaning Step	Traditional ML	Embeddings	Why?
Remove punctuation	Yes	Maybe	Separate it, do not remove
Lowercase	Yes	Maybe	Depends if case matters
Remove stop words	Often	No	Stop words provide context
Stemming	Often	No	Model learns word relations
Remove numbers	Sometimes	Replace	Use <NUM> token

Key Difference

Traditional ML (bag-of-words, TF-IDF): You manually reduce vocabulary because the model cannot learn word relationships.

Embeddings/Neural models: The model learns that "running" $\approx$ "runs" automatically, so stemming just destroys useful information.

Recommended Cleaning for Embeddings

Minimal cleaning approach:

1. **Tokenise carefully** – separate punctuation but keep words intact
2. **Lowercase (maybe)** – depends on whether case matters for your task
3. **Keep stop words** – they provide valuable context
4. **Do NOT stem** – let the model learn word relationships
5. **Handle rare words** – replace very rare words with a special token
6. **Handle numbers** – replace with a placeholder like <NUM>

Example transformation:

- Original: "The cat sat on mat #42 in 2024!"
- Cleaned: "the cat sat on mat <NUM> in <NUM> !"

Key principle: Preserve information that helps the model understand context and meaning.

Quick Decision Guide: What Cleaning Do I Need?

Your Task	Recommended Cleaning
Search engine	Stem both queries and documents; lowercase; remove punctuation; keep original for display
Spam classification (traditional ML)	Lowercase; remove punctuation; remove stop words; stem
Sentiment analysis	Keep punctuation (! matters); keep stop words ("not" matters); maybe lowercase
Training Word2Vec/GloVe	Minimal: lowercase (maybe); separate punctuation; keep stop words; NO stemming
Using BERT/GPT	Usually none! These models have their own tokenisers
Named Entity Recognition	Keep case ("Apple" vs "apple"); keep punctuation

Golden rule: When in doubt, start with minimal cleaning and add more only if it improves your results!

Part 9

Summary

Key Takeaways

1. **Always examine your data first** – understand what you are working with before cleaning
2. **Text cleaning is task-specific** – there is no one-size-fits-all approach
3. **Manual methods** work for simple cases:
   - `split()` for whitespace tokenisation
   - `translate()` for removing punctuation
   - `lower()` for case normalisation
4. **NLTK provides powerful tools**:
   - `word_tokenize()` and `sent_tokenize()`
   - Stop word lists for multiple languages
   - Porter Stemmer and lemmatisers
5. **Word embeddings** represent words as vectors that capture meaning
6. **For embeddings**, use minimal cleaning – let the model learn patterns

Quick Reference: Cleaning Methods

Task	**Python/NLTK**	**Example**
Load text	`open().read()`	Load file into string
Split by whitespace	`text.split()`	"hello world" → ["hello", "world"]
Split by regex	`re.split()`	Splits on non-word chars
Tokenise (NLTK)	`word_tokenize()`	Smarter word splitting
Lowercase	`word.lower()`	"Hello" → "hello"
Remove punctuation	`translate(table)`	"hello!" → "hello"
Remove stop words	Filter with list	Remove "the", "a", etc.
Stemming	`PorterStemmer()`	"running" → "run"

References and Further Reading

Tutorial Source:

- Jason Brownlee, "How to Clean Text for Machine Learning with Python", Machine Learning Mastery
  https://machinelearningmastery.com/clean-text-machine-learning-python/

Word Embeddings:

- Mikolov et al. (2013), "Efficient Estimation of Word Representations in Vector Space" (Word2Vec paper)
- Pennington et al. (2014), "GloVe: Global Vectors for Word Representation"

NLTK Documentation:

- NLTK Official Website: `https://www.nltk.org/`
- NLTK Book (free): `https://www.nltk.org/book/`

Sample Text:

- "Metamorphosis" by Franz Kafka (Project Gutenberg)
  `https://www.gutenberg.org/ebooks/5200`

Questions?

Prof. Georgina Cosma

g.cosma@lboro.ac.uk