Python for Language Processing

(3b) Python for Text Processing

Dr. Jakob Prange

Fakultät für Angewandte Informatik - Universität Augsburg

CL Fall School 24

Credit: This course is based on material developed by
Annemarie Friedrich, Stefan Thater, Michaela Regneri, and Marc Schulder at Saarland University

Python for Text Processing

Examples we have seen so far:

- opening, reading, and writing text files
- splitting strings into words (at whitespaces)
- counting words

Other things that are possible:

- chatbots
- spam detection
- machine translation
- automatic summarization

$\rightarrow$ Machine learning models for processing and generating text

HuggingFace 🤗 Demo:
`https://colab.research.google.com/github/huggingface/notebooks/blob/master/course/en/chapter1/section3.ipynb`

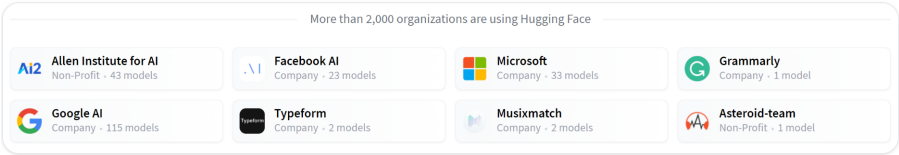

Figure: `https://huggingface.co/learn/nlp-course/chapter1/3`

Python for Text Processing

Text Processing Libraries:

- HuggingFace 
- spaCy
- slightly older: NLTK

Under the hood:

- vectors, tensors (numpy, pytorch)
- tokenization (regex, BPE)
- datasets (pandas, efficient looping & batching)
- processing pipelines

Python for Text Processing

Text Processing Libraries:

- HuggingFace  ← **TODAY**
- spaCy
- slightly older: NLTK

Under the hood:

- vectors, tensors (numpy, pytorch)
- **tokenization (regex, BPE)** ← **TODAY**
- datasets (pandas, efficient looping & batching)
- **processing pipelines** ← **TODAY**

Huggingface Pipelines

Do the quiz! Before answering each question, try to find the answer in the resources below:

- Quiz:
  `https://huggingface.co/learn/nlp-course/chapter2/8`

Resources:

- Transformers:
  `https://huggingface.co/learn/nlp-course/chapter1/3`
- Pipelines:
  `https://huggingface.co/learn/nlp-course/chapter2/2`
- API: `https://huggingface.co/docs/transformers/main_classes/pipelines?search=true`
- And of course the usual suspects: Google, stackoverflow, ...

Review Text Processing Pipelines

Interactive / on the board

Tokenization

- Problem: We have a string of text, but models operate on numbers ($\rightarrow$ vectors)
- Idea: Split up text into pieces
  - Fixed-size set of unique pieces
  - Assign IDs and vectors to each unique piece
  - Use IDs to map between text and vectors
- What should the pieces be?

Tokenization

What should the pieces be?

- Words, of course!
- What is a word?

How many words are in this sentence?

"Building W is far away from building N, but not as far as Alte Uni, isn't it?"

Tokenization

What should the pieces be?

- Words, of course!
- What is a word?

How many words are in this sentence?

```
"Building W is far away from building N, but not as far as Alte Uni, isn't it?"
```

Tokenization

What should the pieces be?

- Words, of course!
- What is a word?

How many words are in this sentence?

```
"Building W is far away from building N, but not as far as Alte Uni, isn't it?"
```

- Between 13 and 21, depending on how you count!

Tokenization

What should the pieces be?

- Words, of course!
- What is a word?

How many words are in these sentences?

姚明进入总决赛

`So jung kema nimma zam!`

Normalization

- `So jung kema nimma zam!` → `So jung kommen wir nicht mehr zusammen!`
  - `kema` → `kommen wir`
  - `nimma` → `nicht mehr`
- Non-standard spellings, typos, mixed languages, ...
  - Normalization!

Language is not deterministic!

How can we search for any of these?

- `"woodchuck"`
- `"woodchucks"`
- `"Woodchuck"`
- `"Woodchucks"`

- `"WOODCHUCKSSSSS"`
- `"wudchugz"`

Language is not deterministic!

How can we search for any of these?

- "woodchuck"
- "woodchucks"
- "Woodchuck"
- "Woodchucks"

- "WOODCHUCKSSSSS"
- "wudchugz"

Language is not deterministic!

How can we search for any of these?

- `"woodchuck"`
- `"woodchucks"`
- `"Woodchuck"`
- `"Woodchucks"`

- `"WOODCHUCKSSSSS"` → RegEx still fine
- `"wudchugz"` → sound-based

Excursus: Pronunciation

```
1  import nltk
2
3  phon_dict = nltk.corpus.cmudict.dict()
4
5  phon_dict['c']       # [['S', 'IY1']]
6  phon_dict['co']      # [['K', 'OW1']]
7  phon_dict['compute'] # [['K', 'AA1', 'M', 'P', 'Y',
8                       #   'UW1', 'T']]
9
10 phon_dict['un']      # [['AH1', 'N'],
11                      #  ['Y', 'UW1', 'EH1', 'N']]
```

Regular Expressions

- What is RegEx?
  - A formal language for specifying sets of text strings.
  - No from-scratch intro to RegEx here
- How to use it in Python?
  - `import re`
  - Go over important methods and string formatting/escaping

The `re` Module

```
1  import re
2  import nltk
3
4
5  wordlist = [w for w in nltk.corpus.words.words('en')
6                 if w.islower()]
7
8  print([w for w in wordlist if re.search('ed$', w)])
9  # ['abaissed', 'abandoned', 'abased', 'abashed',
10 #    'abatised', 'abed', 'aborted', ...]
```

The `re` Module — Important Methods

```
1  import re
2
3  string = 'Woodchucks woodchucks wudchugz'
4  pattern = '[Ww]oodchucks?'
5
6  re.match(pattern, string)    # from beginning
7  re.search(pattern, string)   # anywhere (first)
8  re.findall(pattern, string) # anywhere (all)
9
10 re.split(pattern, string, maxsplit=0)
11
12 repl = 'groundhog'
13 re.sub(pattern, repl, string, count=0) # substitute all
14
15 re.compile(pattern) # compile pattern into object
```

The `re` Module — Important Methods

Better because more readable and much more efficient!

```
1  import re
2
3  string = 'Woodchucks woodchucks wudchugz'
4  pattern = '[Ww]oodchucks?'
5  pattern = re.compile(pattern) # compile first!!
6
7  pattern.search(string)
8  pattern.match(string)
9  pattern.findall(string)
10
11 pattern.split(string, maxsplit=0)
12
13 repl = 'groundhog'
14 pattern.sub(repl, string, count=0)
```

The `re` Module — Examples

```
1  import re
2  import nltk
3
4  chat_words = sorted(
5       set(w for w in nltk.corpus.nps_chat.words())
6  )
7
8  print([w for w in chat_words
9            if re.search('^m+i+n+e+$', w)])
10 # ['miiiiiiiiiiiiinnnnnnnnnnneeeeeeeee',
11 # 'miiiiiinnnnnnnnnneeeeeeee', 'mine', ...]
12
13 print([w for w in chat_words
14            if re.search('^[ha]+$', w)])
15 # ['a', 'aaaaaaaaaaaaaaaa', 'aaahhhh', 'ah', ...,
16 # 'hah', 'haha', 'hahaaa', 'hahah', 'hahaha', ...]
```

The `re` Module — Readings

- https://docs.python.org/3/library/re.html
- https://people.cs.georgetown.edu/nschneid/cosc272/f17/02_py-notes.html
- https://www.nltk.org/book/ch03.html

The `re` Module — Exercise

Write a function `f` that **splits** an input string on commas, semicolons and newlines.

Ignore lines that **begin and end** with **at least two** plus signs.

Each new string resulting from the split should be wrapped in double quotes and printed on a separate line, together with the length of the string.

```
1  s = """The fox; the
2  +++this is a comment++
3  summer is here,,soon?"""
4
5  f(s) # should print the following
6  # "The fox"          len:  7
7  # " the"             len:  4
8  # "summer is here"   len: 14
9  # ""                 len:  0
10 # "soon?"            len:  5
```

Excursus: Raw Strings and Format Strings

```
1 print('normal string\n with \\ backslash \\n')
2 # normal string
3 # with \ backslash \n
4
5 print(r'raw string\n with \ backslash \n')
6 # raw string\n with \ backslash \n
7
8 var = 2; print(f'format string with var {var}')
9 # format string with var 2
```

Excursus: Format Strings

```
1  import math
2  var = 2
3
4  print(f'format string with var {var} and {math.pi}')
5  # format string with var 2 and 3.141592653589793
6
7  print(f'format string with var {var:.2f}'
8        f' and {math.pi:.2f}')
9  # format string with var 2.00 and 3.14
10
11 table = {'Sjoerd': 4127, 'Jack': 4098, 'Saoirse': 767}
12 for name, phone in table.items():
13     print(f'{name:10} ==> {phone:10d}')
14 # Sjoerd     ==>       4127
15 # Jack       ==>       4098
16 # Saoirse    ==>        767
```

Find me all instances of the word “the” in a text.

RegEx Errors

Find me all instances of the word “the” in a text.

- `the`
  Misses capitalized examples
- `[tT]he`
  Incorrectly returns “o**ther**” or “**the**ology”
- `[^a-zA-Z][tT]he[^a-zA-Z]`

Find me all instances of the word “the” in a text.

- `the`
  Misses capitalized examples
- `[tT]he`
  Incorrectly returns “o**the**r” or “**the**ology”
- `[^a-zA-Z][tT]he[^a-zA-Z]`

Find me all instances of the word “the” in a text.

- `the`
  Misses capitalized examples
- `[tT]he`
  Incorrectly returns “o**ther**” or “**the**ology”
- `[^a-zA-Z][tT]he[^a-zA-Z]`

RegEx Errors

The process we just went through was based on fixing two kinds of errors:

1. Matching strings that we should not have matched (there, then, other):
   False positives (Type I errors)
2. Not matching things that we should have matched (The):
   False negatives (Type II errors)

In NLP we are always dealing with these kinds of errors.
Reducing the error rate for an application often involves two opposing efforts:

- Increasing accuracy or precision (minimizing false positives)
- Increasing coverage or recall (minimizing false negatives)

RegEx Summary

- Regular expressions play a surprisingly large role
  - Sophisticated sequences of regular expressions are often the first model for any text processing text
- For hard tasks, we use machine learning classifiers
  - But regular expressions are still used for pre-processing, or as features in the classifiers
  - Can be very useful in capturing generalizations

RegEx Tokenizers

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Figure: Tokenization in NLTK (Bird, Loper and Klein (2009), Natural Language Processing with Python. O'Reilly)

Advanced Tokenizers

- As we saw in the beginning, tokenization is just the first step in a pipeline
- Modern text processing models (the core of the pipeline) are neural networks operating on fixed-size vectors and matrices
- Simple whitespace- and regex-based tokenizers are very flexible but do not keep track of fixed-size vocabularies (sets of word types)
- Solution:
  - Compression algorithms like byte-pair encoding (BPE) learn suitable vocabularies from data
  - These vocabularies can contain single characters and parts of words (subwords)
- The best part is: BPE and advanced variants (SentencePiece, WordPiece, Unigram) are already implemented in HuggingFace

Advanced Tokenizers

- As we saw in the beginning, tokenization is just the first step in a pipeline
- Modern text processing models (the core of the pipeline) are neural networks operating on fixed-size vectors and matrices
- Simple whitespace- and regex-based tokenizers are very flexible but do not keep track of fixed-size vocabularies (sets of word types)
- Solution:
  - Compression algorithms like byte-pair encoding (BPE) learn suitable vocabularies from data
  - These vocabularies can contain single characters and parts of words (subwords)
- The best part is: BPE and advanced variants (SentencePiece, WordPiece, Unigram) are already implemented in HuggingFace

Advanced Tokenizers

- As we saw in the beginning, tokenization is just the first step in a pipeline
- Modern text processing models (the core of the pipeline) are neural networks operating on fixed-size vectors and matrices
- Simple whitespace- and regex-based tokenizers are very flexible but do not keep track of fixed-size vocabularies (sets of word types)
- Solution:
  - Compression algorithms like byte-pair encoding (BPE) learn suitable vocabularies from data
  - These vocabularies can contain single characters and parts of words (subwords)
- The best part is: BPE and advanced variants (SentencePiece, WordPiece, Unigram) are already implemented in HuggingFace

Huggingface Tokenizers (and Models)

```
1  from transformers import AutoTokenizer
2  tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
3  example = "My name is Sylvain."
4
5  encoding = tokenizer(example)
6  print(type(encoding))
7  # <class 'BatchEncoding'>
8
9  encoding.tokens()
10 # ['[CLS]', 'My', 'name', 'is', 'S', '##yl', ...]
11 encoding.word_ids()
12 # [None, 0, 1, 2, 3, 3, ...]
13 start, end = encoding.word_to_chars(3)
14 example[start:end]
15 # Sylvain
16
17 encoding.input_ids
18 # [101, 1422, 1271, 1110, 156, ...]
```

SpaCy Pipelines

`https://spacy.io/models`

Linguistic structure, tagging etc.

Same as with Huggingface, all the heavy-lifting is implemented, you just need to know what to look for and what you want to customize