Week 2: Words & Tokens

POP77032 Quantitative Text Analysis for Social Scientists

Tom Paskhalis

Overview

- Character Encoding
- Words
- Tokens
- Regular Expressions

Character Encoding

Foundations of Computer Memory

- Bits:
  - The smallest unit of digital data.
  - Can be either 0 or 1.
  - n bits can represent 2^n different values.
  - E.g. 2 bits can represent 4 values: 00, 01, 10, 11.
- Bytes:
  - 8 bits = 1 byte
  - Thus, 1 byte can represent 256 values: $[00000000, 00000001, \ldots, 11111111]$.
  - Metric aggregations than are kilobyte (KB), megabyte (MB), gigabyte (GB), etc.

Character Encoding

- **Character** - "the smallest component of written language that has semantic value" (https://unicode.org/glossary/#character).
  - E.g. "h", "ε", "4", "&", "!", "€", "🤖".
- **Character set** - a collection of characters.
  - E.g. Latin alphabet, Greek alphabet, Arabic numerals, punctuation marks, etc.
- **Code point** - the unique value assigned to each character in a set.
  - Depends on what is a considered a valid value: binary - 101101, decimal - 45, hexadecimal - 2D, etc.
- A mapping between code points and characters is called an **encoding**.

ASCII

- **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) - one of the earlier wide-spread character encodings.
- Only encodes $2^7 = 128$ characters (of which 95 are printable, others for teletype).
  - Essentially, English alphabet, Arabic numerals, and some punctuation marks.
- Later extended to $2^8 = 256$ characters
  (aka **ISO-8859-1**, **Latin-1**, closely related to **Windows-1252**).
  - Added support for most Western European languages.
- A lot more needed to support all the world's languages…

ASCII: Code Points

Bits b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4 ↓	b3 ↓	b2 ↓	b1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

(Wikipedia & US DoD)

- E.g., decimal code point for “A” is 65, comprised of these bits:
  - 1000001 (original ASCII)
  - 01000001 (ISO-8859-1)

Unicode

- Designed to support all the world’s writing systems that can be digitized.
- Variable-length, between 1 and 4 bytes (8 and 32 bits).
- First 128 code points are the same as in ASCII (backward compatibility).
- **UTF-8** - most common Unicode encoding (also **UTF-16**, but more rare):
  - 1 byte for ASCII characters.
  - 2 bytes for most Latin, Greek, Cyrillic, CJK, etc.
  - 3 bytes for the rest of the BMP.
  - 4 bytes for the rest of Unicode.
- Supports over 1.1M code points (as of Unicode 17.0 ~160K assigned).

UTF-8: Code Points

Decimal	Binary	Hexadecimal	UTF-8	Character	Description
65	01000001	0x41	U+0041	A	Latin Capital Letter A
66	01000010	0x42	U+0042	B	Latin Capital Letter B
67	01000011	0x43	U+0043	C	Latin Capital Letter C
68	01000100	0x44	U+0044	D	Latin Capital Letter D
69	01000101	0x45	U+0045	E	Latin Capital Letter E
70	01000110	0x46	U+0046	F	Latin Capital Letter F

- E.g. code point for “A”
  - 65 in decimal numeral system
  - 1000001 in binary (original ASCII)
  - 41 in hexadecimal, represented as U+0041 in UTF

Text Encoding: Remarks

- Text encoding provides an *abstract* representation of characters as code points.
- I.e. representing characters as code points is different from their visual rendering.
- The same character (e.g. “A”) can have infinitely many visual representations (fonts, sizes, colors, etc.).
- And some characters (e.g. extinct languages) can have no available **glyphs** (fonts) to render them.
- Inside an actual file we have code points encoded using a specific encoding (e.g. UTF-8).

UTF-8 and Python

- Starting from Python 3 all strings are using Unicode by default.
- I.e. each string is a sequence of Unicode-encoded code points.

```
s = "Hello, 世界!"
len(s)
```

```
10
```

```
for char in s:
    print(f"Character: {char}, Code point: {ord(char)}")
```

```
Character: H, Code point: 72
Character: e, Code point: 101
Character: l, Code point: 108
Character: l, Code point: 108
Character: o, Code point: 111
Character: ,, Code point: 44
Character:  , Code point: 32
Character: 世, Code point: 19990
Character: 界, Code point: 30028
Character: !, Code point: 33
```

UTF-8 and R

- In R strings are also using UTF-8 by default (including Windows from R 4.2.0).

```r
s <- "Hello, 世界!"
nchar(s)
```

```
[1] 10
```

```r
for (char in strsplit(s, "")[[1]]) {
  cat(sprintf("Character: %s, Code point: %d\n", char, utf8ToInt(char)))
}
```

```
Character: H, Code point: 72
Character: e, Code point: 101
Character: l, Code point: 108
Character: l, Code point: 108
Character: o, Code point: 111
Character: ,, Code point: 44
Character:  , Code point: 32
Character: 世, Code point: 19990
Character: 界, Code point: 30028
Character: !, Code point: 33
```

Text Encoding: Things to Try

- Pick a movie you like.
- Go to OpenSubtitles.
- Find subtitles for that movie in a language that uses a different script.
- Download the subtitles and try to open them in a text editor.
- Check the ‘guessed’ encoding of the file.
- Are all characters displayed correctly?
- Try to open the file programmatically in R or Python.

Words and Tokens

How many words in a sentence?

> Hohohoho, Mister Finn, you're going to be Mister Finnagain!

- It depends!
- 9 words if we exclude punctuation and treat 'you're' as a single word.
- 10 words if we exclude punctuation and split 'you're' into 'you' and 're'.
- 12 words if we include punctuation and treat 'you're' as a single word.
- 13 words if we include punctuation and split 'you're' into 'you' and 're'.

How many words in a sentence?

> Hello, world!

And in:

> Hello, 世界!

- Not every written language uses spaces to separate words!
- E.g. Chinese, Japanese, Thai do not.

Vocabulary

- It is important to distinguish:
  - **Word types** - number of unique words in a text (vocabulary size).
  - **Word instances** - total number of words in a text (text length).
- But the number of all word forms can be very large!
- Which means that any model is likely to encounter words that it has not seen during training.
- Thus, instead of actual words, we would use something more flexible, aka **tokens**.

Tokens

- **Token** - an instance of a sequence of characters that are grouped together as a useful semantic unit for processing.
- Some options:
  - Word tokens - sequences of characters separated by spaces/punctuation.
  - Subword tokens - smaller units than words (e.g. syllables, morphemes).
  - Character tokens - individual characters.
- Different tasks would require different **tokenization** strategies.
- In social science applications word-derived tokens are most common.
- In NLP applications subword tokens are often used (e.g. Byte Pair Encoding).

Regular Expressions

Find and Replace

- Basic *find* and *replace* operations are the most widely available text manipulation tools.
- They are available in text editors, word processors, IDEs, etc.
- Oftentimes they are used to identify exact matches.
- But what if we want to find words that aren’t exactly the same?

How to find a word?

Imagine that you want to identify all instances of the word 'times' in the following text:

> It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair…

- This is straightforward:

```python
import re
```

```python
tale = """It was the best of times, it was the worst of times,
it was the age of wisdom, it was the age of foolishness,
it was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair..."""
```

```python
# r before quotation mark denotes a raw string
re.findall(r"times", tale)
```

```
['times', 'times']
```

How to find a word?

- Now, say, you want to find all instances of ‘it’.
- This is a bit trickier since ‘it’ can be both capitalised and lowercase.
- To match both *I* and *i* we can use a **character set**: `[Ii]`
- This will match either ‘I’ or ‘i’.
- Also knows as **character disjunction**

```
its = re.findall(r"[Ii]t", tale)
```

```
len(its)
```

```
11
```

```
its
```

```
['It', 'it', 'it', 'it', 'it', 'it', 'it', 'it', 'it', 'it', 'it']
```

Regular Expressions

- What we have seen above is an example of a **regular expression**,
- A sequence of characters that define a search pattern.
- Also knows as **regex** or **regexp**.
- This is, probably, the single most widely used tool for text processing.
- It is supported in most programming languages and text editors.

Regular Expressions: Main Patterns

<table>
<tr><th>Regex</th><th>Name</th><th>Description</th><th>Example</th><th>Matches</th></tr>
<tr><td>.</td><td>Wildcard</td><td>Matches any single character (except newline, usually)</td><td>c.t</td><td>cat | cut</td></tr>
<tr><td>*</td><td>Zero or more</td><td>Matches 0 or more of the preceding token</td><td>lo*l</td><td>ll | lol | looool</td></tr>
<tr><td>+</td><td>One or more</td><td>Matches 1 or more of the preceding token</td><td>lo+l</td><td>lol | looool</td></tr>
<tr><td>?</td><td>Optional</td><td>Matches 0 or 1 of the preceding token</td><td>colou?r</td><td>color | colour</td></tr>
<tr><td>{n}</td><td>Exact count</td><td>Matches exactly n occurrences</td><td>a{3}</td><td>aaa</td></tr>
<tr><td>{n,}</td><td>At least n</td><td>Matches n or more occurrences</td><td>a{2,}</td><td>aa | aaa</td></tr>
<tr><td>{n,m}</td><td>Range</td><td>Matches between n and m occurrences</td><td>a{2,4}</td><td>aa | aaa | aaaa</td></tr>
<tr><td>^</td><td>Start anchor</td><td>Matches start of string</td><td>^Hello</td><td>Hello world</td></tr>
<tr><td>$</td><td>End anchor</td><td>Matches end of string</td><td>world$</td><td>Hello world</td></tr>
<tr><td>[]</td><td>Character class</td><td>Matches any one character inside brackets</td><td>[aeiou]</td><td>a | e | i</td></tr>
<tr><td>[^]</td><td>Negated class</td><td>Matches any character not in brackets</td><td>[^0-9]</td><td>a | #</td></tr>
<tr><td>|</td><td>Alternation</td><td>Logical OR</td><td>cat|dog</td><td>cat | dog</td></tr>
<tr><td>()</td><td>Grouping</td><td>Groups tokens and captures matches</td><td>(ab)+</td><td>ab | abab</td></tr>
</table>

Regular Expressions: Example

Going back to the original quote, let's find all the attributes of the period that Charles Dickens describes:

```
tale
```

```
'It was the best of times, it was the worst of times,\nit was the age of wisdom, it was the age of
foolishness,\nit was the epoch of belief, it was the epoch of incredulity,\nit was the season of Light, it was
the season of Darkness,\nit was the spring of hope, it was the winter of despair...'
```

We could start by constructing a regex part that captures *age*, *epoch* and *season*.

```
period = r"(age|epoch|season)"
```

Since we don't want to extract (capture) these words as such, we will re-write it as a **non-capturing group**:

```
# Note the ?: prefix
period = r"(?:age|epoch|season)"
```

Now we will add the following *of* to the regex with `\s+` matching one or more spaces:

```
periodof = rf"{period}\s+of\s+"
periodof
```

```
'(?:age|epoch|season)\\s+of\\s+'
```

Regular Expressions: Example Continued

Finally, we can add the search pattern for the actual word with the attribute of the period:

```
# (\w+) matches a sequence of word characters (letters, digits, or underscores)
attributes = rf"{periodof}(\w+)"
attributes
```

```
'(?:age|epoch|season)\\s+of\\s+(\\w+)'
```

Now it’s time to apply it:

```
re.findall(attributes, tale)
```

```
['wisdom', 'foolishness', 'belief', 'incredulity', 'Light', 'Darkness']
```

Regular Expressions: Remarks

- The construction of regular expressions used to be a very laborious process.
- The advancement of gen AI vastly simplified it.
- However, you do still need to understand the basic building blocks and syntax.
- Guiding the model with clear instructions and examples is crucial.

Extra

Python documentation on regular expressions

Extra

R documentation on regular expressions

Next

- Tutorial: Regular expressions and string distances
- Next week: Quantifying Texts
- Assignment 1: Due 15:59 on Wednesday, 11th February (submission on Blackboard)