Processing Raw Text
POS Tagging

Florian Fink
- Folien von Desislava Zhekova -

CIS, LMU
finkf@cis.lmu.de

January 26, 2021

Outline

Dealing with other formats

Often enough, content on the Internet as well as locally stored content is transformed to a number of formats different from plain text (`.txt`).

- RTF – Rich Text Format (`.rtf`)
- HTML – HyperText Markup Language (`.html, .htm`)
- XHTML – Extensible HyperText Markup Language (`.xhtml, .xht, .xml, .html, .htm`)
- XML – Extensible Markup Language (`.xml`)
- RSS – Rich Site Summary (`.rss, .xml`)

Dealing with other formats

Additionally, often text is stored in binary formats, such as:

- MS Office formats – (`.doc`, `.dot`, `.docx`, `.docm`, `.dotx`, `.dotm`, `.xls`, `.xlt`, `.xlm`, `.ppt`, `.pps`, `.pptx` ... and many others)
- PDF – Portable Document Format (`.pdf`)
- OpenOffice formats – (`.odt`, `.ott`, `.oth`, `.odm` ... and others)

HTML

`https://en.wikipedia.org/wiki/Python_(programming_language)'`

```python
import urllib
url="https://en.wikipedia.org/wiki/Python_(programming_language)"
with urllib.request.urlopen(url) as response:
    print(response.info())
# prints
# ...
# Content-Type: text/html; charset=UTF-8
# Content-Length: 470402
# ...
    html = response.read().decode("utf-8")
    print(html)
# prints
#'<!DOCTYPE html>
#<html class="client-nojs">
#<head>
#<meta charset="UTF-8"/>
#<title>Python (programming language) - Wikipedia</title>
```

HTML

HTML is often helpful since it marks up the distinct parts of the document, which makes them easy to find:

```
<title> Python (programming language) — Wikipedia</title>
...
```

Beautiful Soup

- Python library for pulling data out of HTML and XML files.
- can navigate, search, and modify the parse tree.

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story"> ... </p>
"""
```

Beautiful Soup

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

Beautiful Soup

BeautifulSoup object represents the document as a nested data structure:

```python
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
print(soup.prettify())
# <html>
#  <head>
#   <title>
#    The Dormouse's story
#   </title>
#  </head>
#  <body>
#   <p class="title">
#    <b>
#     The Dormouse's story
#    </b>
#    ...
```

Beautiful Soup

Simple ways to navigate that data structure: say the name of the tag you want

```
soup.title
# <title>The Dormouse's story</title>

soup.title.string
# u'The Dormouse's story'

soup.title.parent.name
# u'head'

soup.p
# <p class="title"><b>The Dormouse's story</b></p>

soup.p['class']
# u'title'
```

Beautiful Soup

Simple ways to navigate that data structure:

```
soup.a
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.find(id="link3")
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

Beautiful Soup

One common task is extracting all the URLs found within a page's <a> tags:

```python
for link in soup.find_all('a'):
    print(link.get('href'))
# http://example.com/elsie
# http://example.com/lacie
# http://example.com/tillie
```

Beautiful Soup

Another common task is extracting all the text from a page:

```python
print(soup.get_text())
# The Dormouse's story
#
# The Dormouse's story
#
# Once upon a time there were three little sisters; and their names were
# Elsie,
# Lacie and
# Tillie;
# and they lived at the bottom of a well.
#
# ...
```

Beautiful Soup

Installing Beautiful Soup:

- `apt-get install python3-bs4 (for Python 3)`
- `pip install bs4`

Binary formats

Nowadays we often store text in formats that are not human-readable: e.g. binary format (e.g. `.doc`, `.pdf`). These formats are not as easily processed as simple text.

Binary formats

There are a number of modules that can be installed and used for extracting data from binary files. Yet, depending on the files, the output is not always clean and easily usable.

Binary formats

```
import nltk
import PyPDF2

with open("text.pdf", "rb") as f:
    pdf = PyPDF2.PdfFileReader(f)

for page in pdf.pages:
    print(page.extractText())

# prints each of the pages as a raw text.
```

Binary formats

Snippet from a pdf document "intro.pdf"

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CENTRUM FÜR INFORMATIONS- UND SPRACHVERARBEITUNG
STUDIENGANG COMPUTERLINGUISTIK

Symbolische Programmiersprache

Abstract This course will use the Python programming language as the basis for various computational linguistic implementations. We will cover a wide range of natural language processing (NLP) tasks, such as tokenization, keyword extraction, normalization and stemming, categorization and tagging, as well as classification, chunking and language identification. All the latter are basic NLP tasks that will be discussed and their implementation in Python will be realized during the practical exercise in connection to the course. With respect to each task, we will concentrate on the problems that this task faces and the possible solutions to them within the Python framework. All students will be required to complete weekly assignments and write a term paper (10-12 pages) as a summary of the discussed topics and their importance and application for computational linguistics.

Binary formats

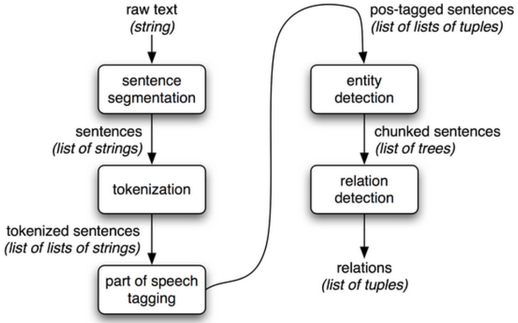
```python
import nltk
import PyPDF2

with open("text.pdf", "rb") as f:
    pdf = PyPDF2.PdfFileReader(f)

for page in pdf.pages:
    print(page.extractText()+"\n")
```

Binary formats

The full text might be extracted, but not in a easily usable format as here:

```
SymbolischeProgrammierspracheAbstractThiscoursewillusethePythonprogramminglanguagea
sthebasisforvariouscomputationallinguisticimplementations.Wewillcoverawiderangeofna
turallanguageprocessing(NLP)tasks,suchastokenization,keywordextraction,normalizatio
nandstemming,categorizationandtagging,aswellaschunkingandlanguageidenAllthelatterar
ebasicNLPtasksthatwillbediscussedandtheirimplementationinPythonwillberealizedduring
thepracticalexerciseinconnectiontothecourse.Withrespecttoeachtask,wewillconcentrate
ontheproblemsthatthistaskfacesandthepossiblesolutionstothemwithinthePythonframework
.Allstudentswillberequiredtocompleteweeklyassignmentsandwriteatermpaper(10-12pages)
asasummaryofthediscussedtopicsandtheirimportanceandapplicationforcomputationallingu
istics.FormatofthecourseCredits:4SWS(6ECTS)Coursetimes:Tuesdays16:00{18:00,Thursday
s16:00{18:00Location:Tuesdays{RoomL155andThursdays{CIP-PoolAntarktis30Sessions:14.1
0.2013{07.02.2014ThecoursewillbeheldinGermanandEnglish.Coursewebpage:http://www.cis
.uni-muenchen.de/kurse/desi/spInstructor:DesislavaZhekovaContact:desi@cis.uni-muenc
hen.deC106Hours:Wednesdays10:00{11:00,butfeelfreetocomebyanytime.Anemailinadvancewi
llmakesurethatyouactuallyme!
```

NLP pipeline



POS Tagging Overview

- **parts-of-speech** (word classes, lexical categories, POS) – e.g. verbs, nouns, adjectives, etc.
- **part-of-speech tagging** (POS tagging, tagging) – labeling words according to their POS
- **tagset** – the collection of tags used for a particular task

Using a Tagger

A part-of-speech tagger, or POS tagger, processes a sequence of words, and attaches a part of speech tag to each word:

```
import nltk

text = nltk.word_tokenize("And now for something
    completely different")
print(nltk.pos_tag(text))

# [('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('
    something', 'NN'), ('completely', 'RB'), ('
    different', 'JJ')]
```

Variation in Tags

```
# [('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')]
```

- CC – coordinating conjunction
- RB – adverb
- IN – preposition
- NN – noun
- JJ – adjective

Documentation

NLTK provides documentation for each tag, which can be queried using the tag, e.g:

```
>>> nltk.help.upenn_tagset('NN')
NN: noun, common, singular or mass
    common-carrier cabbage knuckle-duster Casino
        afghan shed thermostat investment slide
        humour falloff slick wind hyena override
        subhumanity machinist ...
>>> nltk.help.upenn_tagset('CC')
CC: conjunction, coordinating
    & and both but either et for less minus neither
        nor or plus so therefore times v. versus vs.
        whether yet
```

Documentation

Note!

Some POS tags denote variation of the same word type, e.g. NN, NNS, NNP, NNPS, such can be looked up via regular expressions.

```
>>> nltk.help.upenn_tagset('NN*')
NN: noun, common, singular or mass
    common-carrier cabbage knuckle-duster Casino ...
NNP: noun, proper, singular
    Motown Venneboerger Czestochwa Ranzer Conchita
        ...
NNPS: noun, proper, plural
    Americans Americas Amharas Amityvilles ...
NNS: noun, common, plural
    undergraduates scotches bric-a-brac ...
```

Disambiguation

POS tagging does not always provide the same label for a given word, but decides on the correct label for the specific context – disambiguates across the word classes.

```python
import nltk

text = nltk.word_tokenize("They refUSE to permit us to obtain the REFuse permit")
print(nltk.pos_tag(text))

# [('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'), ('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]
```

Example from Brown

Whenever a corpus contains tagged text, the NLTK corpus interface will have a `tagged_words()` method.

```
>>> nltk.corpus.brown.words()
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said',
    ... ]

>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ... ]
```

Variation across Tagsets

Even for one language, POS tagsets may differ considerably!

Variation across Tagsets

Alphabetical list of part-of-speech tags used in the Penn Treebank Project:

Number	Tag	Description
1.	CC	Coordinating conjunction
2.	CD	Cardinal number
3.	DT	Determiner
4.	EX	Existential *there*
5.	FW	Foreign word
6.	IN	Preposition or subordinating conjunction
7.	JJ	Adjective
8.	JJR	Adjective, comparative
9.	JJS	Adjective, superlative
10.	LS	List item marker
11.	MD	Modal
12.	NN	Noun, singular or mass

Variation across Tagsets

The Open Xerox English POS tagset:

Tag	Description	Example
+ADJ	(basic) adjective	[a] blue [book], [he is] big
+ADJCMP	comparative adjective	[he is] bigger, [a] better [question]
+ADJING	adjectival ing-form	[the] working [men]
+ADJPAP	adjectival past participle	[a] locked [door]
+ADJPRON	pronoun (with determiner) or adjective	[the] same; [the] other [way]
+ADJSUP	superlative adjective	[he is the] biggest; [the] best [cake]
+ADV	(basic) adverb	today, quickly
+ADVCMP	comparative adverb	sooner
+ADVSUP	superlative adverb	soonest
+CARD	cardinal (except *one*)	two, 123, IV
+CARDONE	cardinal one	[at] one [time] ; one [dollar]
+CM	comma	,
+COADV	coordination adverbs *either, neither*	either [by law o by force]; [he didn't come] either
+COORD	coordinating conjunction	and, or

Variation across Tagsets

The variation across tagsets is based on the different decisions and the information needed to be included:

- morphologically rich tags
- morphologically poor ones

Arabic Example

For example, in Arabic the morphologically-poor tag `NN` may be divided into the following morphologically-rich variants:

```
(ABBREV NN)
(LATIN NN)
(DET+NOUN NN)
(DET+NOUN+NSUFF_FEM_SG NN)
(NOUN NN)
(NOUN+NSUFF_FEM_SG NN)
(NOUN+NSUFF_MASC_SG_ACC_INDEF NN)
(DEM+NOUN NN)
(DET+NOUN+CASE_DEF_ACC NN)
(DET+NOUN+CASE_DEF_GEN NN)
(DET+NOUN+CASE_DEF_NOM NN)
(DET+NOUN+NSUFF_FEM_SG+CASE_DEF_ACC NN)
(DET+NOUN+NSUFF_FEM_SG+CASE_DEF_GEN NN)
(DET+NOUN+NSUFF_FEM_SG+CASE_DEF_NOM NN)
 (NOUN+CASE_DEF_ACC NN)
 (NOUN+CASE_DEF_GEN NN)
 (NOUN+CASE_DEF_NOM NN)
 (NOUN+CASE_INDEF_ACC NN)
 (NOUN+CASE_INDEF_GEN NN)
 (NOUN+CASE_INDEF_NOM NN)
 (NOUN+NSUFF_FEM_SG+CASE_DEF_ACC NN)
 (NOUN+NSUFF_FEM_SG+CASE_DEF_GEN NN)
 (NOUN+NSUFF_FEM_SG+CASE_DEF_NOM NN)
 (NOUN+NSUFF_FEM_SG+CASE_INDEF_ACC NN)
 (NOUN+NSUFF_FEM_SG+CASE_INDEF_GEN NN)
 (NOUN+NSUFF_FEM_SG+CASE_INDEF_NOM NN)
 (NEG_PART+NOUN NN)
```

NLTK and simplified tags

NLTK includes built-in mapping to a simplified tagset for most complex tagsets included in it:

```
>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ... ]

>>> nltk.corpus.brown.tagged_words(tagset='universal')
[('The', 'DET'), ('Fulton', 'NOUN'), ... ]
```

NLTK and simplified tags

The Universal Part-of-Speech Tagset of NLTK:

Tag	Meaning	English Examples
ADJ	adjective	*new, good, high, special, big, local*
ADP	adposition	*on, of, at, with, by, into, under*
ADV	adverb	*really, already, still, early, now*
CONJ	conjunction	*and, or, but, if, while, although*
DET	determiner, article	*the, a, some, most, every, no, which*
NOUN	noun	*year, home, costs, time, Africa*
NUM	numeral	*twenty-four, fourth, 1991, 14:24*
PRT	particle	*at, on, out, over per, that, up, with*
PRON	pronoun	*he, their, her, its, my, I, us*
VERB	verb	*is, say, told, given, playing, would*
.	punctuation marks	*. , ; !*
X	other	*ersatz, esprit, dunno, gr8, univeristy*

Tagged Corpora for Other Languages

Tagged corpora for several other languages are distributed with NLTK, including Chinese, Hindi, Portuguese, Spanish, Dutch, and Catalan.

```
>>> nltk.corpus.sinica_treebank.tagged_words()
>>> nltk.corpus.indian.tagged_words()
```

```
[('一', 'Neu'), ('友情', 'Nad'), ('嘉珍', 'Nba'), ...]
[('মহিষের', 'NN'), ('সন্তান', 'NN'), (':', 'SYM'), ...]
```

Frequency Distributions of POS Tags

We have calculated Frequency Distributions based on a sequence of words. Thus, we can do so for POS tags as well.

```
import nltk
from nltk.corpus import brown

brown_news_tagged = brown.tagged_words(categories='news',
    tagset='universal')
tag_fd = nltk.FreqDist(tag for (word, tag) in
    brown_news_tagged)
print(tag_fd.most_common())
# [('NOUN', 30640), ('VERB', 14399), ('ADP', 12355), ('.',
    11928), ('DET', 11389), ('ADJ', 6706), ('ADV', 3349),
    ('CONJ', 2717), ('PRON', 2535), ('PRT', 2264), ('NUM',
    2166), ('X', 106)]
```

Example Explorations

```
1 import nltk
2 wsj = nltk.corpus.treebank.tagged_words(tagset='universal')
3 cfd1 = nltk.ConditionalFreqDist(wsj)
4 print(list(cfd1['yield'].keys()))
5 print(list(cfd1['cut'].keys()))
```

???

What is calculated in the lines 4 and 5?

Example Explorations

We can reverse the order of the pairs, so that the tags are the conditions, and the words are the events. Now we can see likely words for a given tag:

```
import nltk

wsj = nltk.corpus.treebank.tagged_words(tagset='universal')
cfd2 = nltk.ConditionalFreqDist((tag, word) for (word, tag)
    in wsj)
print(list(cfd2['VERB'].keys()))

# ['sue', 'leaving', 'discharge', 'posing', 'redistributing
    ', 'emerges', 'anticipates', 'Hold', 'purrs', 'telling
    ', 'obtained', 'ringing', 'mind', ... ]
```

Example Explorations

```
import nltk
from nltk.corpus import brown

brown_news_tagged = brown.tagged_words(categories='news', tagset='universal')
data = nltk.ConditionalFreqDist((word.lower(), tag) for (word, tag) in brown_news_tagged)

for word in data.conditions():
    if len(data[word]) > 3:
        x = data[word].keys()
        print (word, ' '.join(x))
```

???

What is calculated here?

TreeTagger

- The TreeTagger is a tool for annotating text with part-of-speech and lemma information
- is used to tag German, English, French, Italian, Danish, Dutch, Spanish, Bulgarian, Russian, Portuguese, Galician, Greek, Chinese, Swahili, Slovak, Slovenian, Latin, Estonian, etc.
- Sample output:

word	**pos**	**lemma**
The	DT	the
TreeTagger	NP	TreeTagger
is	VBZ	be
easy	JJ	easy
to	TO	to
use	VB	use

TreeTagger

- Download the files from `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/`
- Run the installation script: `sh install-tagger.sh`
- Test it:

```
echo 'Das ist ein gutes Beispiel!' | cmd/tree-tagger-german

        reading parameters ...
        tagging ...
         finished.
das     PDS     die
ist     VAFIN   sein
ein     ART     eine
gutes   ADJA    gut
Beispiel        NN      Beispiel
!       $.      !
```

Default Tagger - Baseline

Baseline approaches in Computational Linguistics are the simplest means to solve the task even if this is connected to a very low overall performance. Baseline approaches still aim at good performance, but the emphasis is put on simplicity and unreliability on other resources.

Default Tagger - Baseline

???

Given a large body of text, what could be the baseline tagging approach that will enable you to easily tag the text without any other resources, tools, knowledge?

Default Tagger - Baseline

In case annotated corpora of the same type is available, one can estimate the most often seen POS tag in it:

```
import nltk
from nltk.corpus import brown

tags = [tag for (word, tag) in brown.tagged_words(categories='news')]
print(nltk.FreqDist(tags).max())

# NN
```

Default Tagger - Baseline

Use the Default Tagger to tag in a baseline mode:

```python
import nltk
from nltk.corpus import brown

raw = 'I do not like green eggs and ham, I do not like them Sam I am!'
tokens = nltk.word_tokenize(raw)
default_tagger = nltk.DefaultTagger('NN')
print(default_tagger.tag(tokens))

# [('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('green', 'NN'), ('eggs', 'NN'), ('and', 'NN'), ('ham', 'NN'), (',', 'NN'), ('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('them', 'NN'), ('Sam', 'NN'), ('I', 'NN'), ('am', 'NN'), ('!', 'NN')]
```

Default Tagger - Baseline

Unsurprisingly, this method performs rather poorly.

```
>>> default_tagger.evaluate(brown.tagged_sents())
0.13130472824476916
```

Regular Expression Tagger

The regular expression tagger assigns tags to tokens on the basis of matching patterns:

```
>>> patterns = [
...   (r'.*ing$', 'VBG'), # gerunds
...   (r'.*ed$', 'VBD'), # simple past
...   (r'.*es$', 'VBZ'), # 3rd singular present
...   (r'.*ould$', 'MD'), # modals
...   (r".*'s$", 'NN$'), # possessive nouns
...   (r'.*s$', 'NNS'), # plural nouns
...   (r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # cardinal numbers
...   (r'.*', 'NN') # nouns (default)
...   ]
```

Note!

These patterns are processed in order, and the first one that matches is applied.

Regular Expression Tagger

```
brown_sents = brown.sents(categories='news')
regexp_tagger = nltk.RegexpTagger(patterns)
print(regexp_tagger.tag(brown.sents()[3]))

 # [('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative', 'NN'), ('handful', 'NN'), ('of', 'NN'), ('such', 'NN'), ('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'), ("''", 'NN'), (',', 'NN'), ('the', 'NN'), ('jury', 'NN'), ('said', 'NN'), (',', 'NN'), ('``', 'NN'), ('considering', 'VBG'), ('the', 'NN'), ('widespread', 'NN'), ('interest', 'NN'), ('in', 'NN'), ('the', 'NN'), ('election', 'NN'), (',', 'NN'), ('the', 'NN'), ('number', 'NN'), ('of', 'NN'), ('voters', 'NNS'), ('and', 'NN'), ('the', 'NN'), ('size', 'NN'), ('of', 'NN'), ('this', 'NNS'), ('city', 'NN'), ("''", 'NN'), ('.', 'NN')]
```

Regular Expression Tagger

Evaluating the Regular Expression Tagger shows that:

```
>>> regexp_tagger.evaluate(brown.tagged_sents())
0.20326391789486245
```

However, as you see, not this efficient! What other possibilities do we have?

Lookup Tagger

A lot of high-frequency words do not have the `NN` tag. Let's find the hundred most frequent words and store their most likely tag. We can then use this information as the model for a "lookup tagger" (an NLTK `UnigramTagger`):

```python
import nltk
from nltk.corpus import brown

fd = nltk.FreqDist(brown.words(categories='news'))
cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
most_freq_words = fd.most_common(100)
likely_tags = dict((word, cfd[word].max()) for (word, _) in most_freq_words)
baseline_tagger = nltk.UnigramTagger(model=likely_tags)
sent = brown.sents(categories='news')[3]
print(baseline_tagger.tag(sent))
```

UnigramTagger

```python
# [('``', '``'), ('Only', None), ('a', 'AT'), ('relative', None), ('handful', None), ('of', 'IN'), ('such', None), ('reports', None), ('was', 'BEDZ'), ('received', None), ("''", "''"), (',', ','), ('the', 'AT'), ('jury', None), ('said', 'VBD'), (',', ','), ('``', '``'), ('considering', None), ('the', 'AT'), ('widespread', None), ('interest', None), ('in', 'IN'), ('the', 'AT'), ('election', None), (',', ','), ('the', 'AT'), ('number', None), ('of', 'IN'), ('voters', None), ('and', 'CC'), ('the', 'AT'), ('size', None), ('of', 'IN'), ('this', 'DT'), ('city', None), ("''", "''"), ('.', '.')]

print(baseline_tagger.evaluate(brown.tagged_sents()))
# 0.46934270990499416
```

Many words have been assigned a tag of `None`, because they were not among the 100 most frequent words. In these cases we would like to assign the default tag of `NN` – process known as **backoff**.

Backoff

```python
import nltk
from nltk.corpus import brown

fd = nltk.FreqDist(brown.words(categories='news'))
cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
most_freq_words = fd.most_common(100)
likely_tags = dict((word, cfd[word].max()) for (word, _) in most_freq_words)
baseline_tagger = nltk.UnigramTagger(model=likely_tags, backoff=nltk.DefaultTagger('NN'))
sent = brown.sents(categories='news')[3]
print(baseline_tagger.tag(sent))
```

Backoff

```python
# [('``', '``'), ('Only', 'NN'), ('a', 'AT'), ('relative', 'NN'), ('handful', 'NN'), ('of', 'IN'), ('such', 'NN'), ('reports', 'NN'), ('was', 'BEDZ'), ('received', 'NN'), ("''", "''"), (',', ','), ('the', 'AT'), ('jury', 'NN'), ('said', 'VBD'), (',', ','), ('``', '``'), ('considering', 'NN'), ('the', 'AT'), ('widespread', 'NN'), ('interest', 'NN'), ('in', 'IN'), ('the', 'AT'), ('election', 'NN'), (',', ','), ('the', 'AT'), ('number', 'NN'), ('of', 'IN'), ('voters', 'NN'), ('and', 'CC'), ('the', 'AT'), ('size', 'NN'), ('of', 'IN'), ('this', 'DT'), ('city', 'NN'), ("''", "''"), ('.', '.')]

print(baseline_tagger.evaluate(brown.tagged_sents()))
# 0.5980888604124038
```

Evaluation overview

tagger	Accuracy
`DefaultTagger('NN')`	0.13
`RegexpTagger(patterns)`	0.20
`UnigramTagger(model)`	0.47
`UnigramTagger(model, backoff)`	0.60

General N-Gram Tagging

The problem of unigram tagging – assigns one tag irrespective of its context:

- `the wind`
- `to wind`

General N-Gram Tagging

`hoping for the` **wind** `to stop blowing`

- **unigram tagging** – one item of context: `wind`

General N-Gram Tagging

`hoping for the` **wind** `to stop blowing`

- **unigram tagging** – one item of context: `wind`
- **bigram tagging** – two items of context: `the wind`

General N-Gram Tagging

`hoping for the` **wind** `to stop blowing`

- **unigram tagging** – one item of context: `wind`
- **bigram tagging** – two items of context: `the wind`
- **trigram tagging** – three items of context: `for the wind`

General N-Gram Tagging

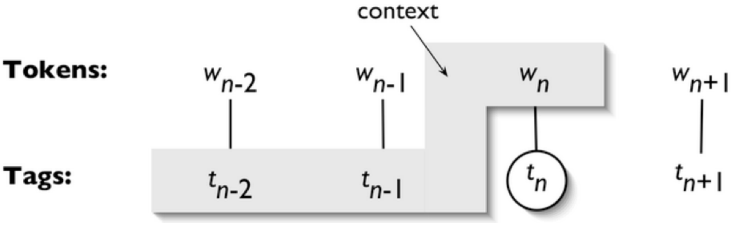
`hoping for the` **wind** `to stop blowing`

- **unigram tagging** – one item of context: `wind`
- **bigram tagging** – two items of context: `the wind`
- **trigram tagging** – three items of context: `for the wind`
- **n-gram tagging** – *n* items of context

General N-Gram Tagging

Note!

In tagging, preceding tokens are only represented by their POS tags!



Lookup Tagger

???

With respect to the data used to train/test the Lookup Tagger in this example, there is a small logical problem. Can you figure out what that problem is?

```
fd = nltk.FreqDist(brown.words(categories='news'))
cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
most_freq_words = fd.most_common(100)
likely_tags = dict((word, cfd[word].max()) for word in most_freq_words)
baseline_tagger = nltk.UnigramTagger(model=likely_tags)
baseline_tagger.evaluate(brown.tagged_sents())
```

Lookup Tagger

A better way to use the data is:

```
size = int(len(brown_tagged_sents) * 0.9)

train_sents = brown_tagged_sents[:size]
test_sents = brown_tagged_sents[size:]

unigram_tagger = nltk.UnigramTagger(train_sents)
unigram_tagger.evaluate(test_sents)

#0.81202033290142528
```

Data Sets

Note!

Not only do we need to separate training and test set from each other, but there are a number other issues that we need to keep in mind:

- The larger the training data is, the better the system is trained – more data beats a cleverer algorithm.
- If the test set is too small, it will not provide an objective evaluation.
- Select training data that is representative for the problem – if you test on the news category, train on this type of data as well.

Data Sets

So, we have a number of different datasets that are used in Machine Learning:

- **training data** – a large number of examples for which the correct answers are already provided and which can be used to train a predictive model. In this case the training process involves inspecting the tag of each word and storing the most likely tag for the 100 most often seen words in it.
- **test data** – a set of data that the system needs to label, which is used to evaluate its performance.
- **development data** – a set of data used as “test set” during system development

Tagging Development

Developing a tagger (similar to developing most other NLP tools) is an iterative process:

1. Implement a base version
2. Train
3. Test (use development data)
4. Analyze errors
5. Implement improvements – optimize
6. Go back to step 2
7. ...
8. Test optimized version (use test data)

Tagging Development

```python
size = int(len(brown.tagged_sents()) * 0.9)
train_sents = brown.tagged_sents()[:size]
dev_sents = brown.tagged_sents()[size:]

t0 = nltk.DefaultTagger('NN')
print(t0.evaluate(dev_sents))
# 0.10674545797447664

t1 = nltk.UnigramTagger(train_sents, backoff=t0)
print(t1.evaluate(dev_sents))
# 0.8914838331133044

t2 = nltk.BigramTagger(train_sents, backoff=t1)
print(t2.evaluate(dev_sents))
# 0.9128371157352109
```

Storing a tagger

Once a final version (an optimized tagger) is developed, it is good to store the tagger. Additionally, training a tagger on a large corpus may take a significant time. Solution – store the tagger (requires the `pickle` module):

```
from pickle import dump
with open('t2.pkl', 'wb') as output:
    dump(t2, output)
```

```
from pickle import load
with open('t2.pkl', 'rb') as input:
    tagger = load(input)
```

Tagging Development

Developing a tagger (similar to developing most other NLP tools) is an iterative process:

1. Implement a base version
2. Train
3. Test (use development data)
4. Analyze errors
5. Implement improvements – optimize
6. Go back to step 2
7. ...
8. Test optimized version (use test data)

But how to analyze the errors?

Optimization

Analyze a summary of the performance

Confusion matrix (simplified version):

```
       |                 N           P                   |
       |           N     U           R     T     U       |
       |     N     P     M     P     O     O     H     V |
-------|-------------------------------------------------+
     N |<12256>   12     .     .     .     .     .    58 |
    NP |    18 <2531>    .     .     .     .     .     . |
   NUM |     .     .  <823>    .     2     .     .     . |
     P |     2     .     . <5817>    .   519     .     . |
   PRO |     .     .    19     . <2931>    .     .     . |
    TO |     .     .     .    44     .  <910>    .     . |
    UH |     .     .     .     .     .     .   <9>     2 |
     V |    61     .     .     .     .     .     . <5111>|
-------+-------------------------------------------------+
(row = reference (correct); col = test (given))
```

Optimization

Creating a confusion matrix:

```
size = int(len(brown.tagged_sents()) * 0.9)
train_sents = brown.tagged_sents(simplify_tags=True)[:size]
test_sents = brown.tagged_sents(simplify_tags=True)[size:]

t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_sents, backoff=t0)
t2 = nltk.BigramTagger(train_sents, backoff=t1)

test = [tag for sent in brown.sents(categories='editorial') for (
    word, tag) in t2.tag(sent)]
gold = [tag for (word, tag) in brown.tagged_words(categories='
    editorial', simplify_tags=True)]
print(nltk.ConfusionMatrix(gold, test))
```

Optimization

Optimizing a tagger would mean that a better solution needs to be provided for ambiguous cases. This could be achieved by:

- more training data
- looking at a wider context - increasing n
- based on the error analysis, e.g. confused labels

Important Concepts

- Baseline approaches
- Optimize the tagger using training and development data:
  - more training data
  - looking at a wider context - increasing *n*
  - based on the error analysis, e.g. confused labels
- Test optimized version on the test data
- Store the tagger

References

- `http://www.nltk.org/book/`
- `https://github.com/nltk/nltk`