Practical assignment 2

Benoît Sagot
Inria (ALMAnaCH)

MVA — Speech and Language Processing — Class #6 — 4th March, 2019

Goal

- **Develop a basic probabilistic constituency parser for French that is based on the CYK algorithm and the PCFG model and that is robust to unknown words**

- Two parts in this introductory talk to the assignment
  - The Levenshtein edit distance algorithm (and a few words on spelling correction in general)
  - The CYK algorithm
- Both are based on dynamic programming, and share underlying ideas

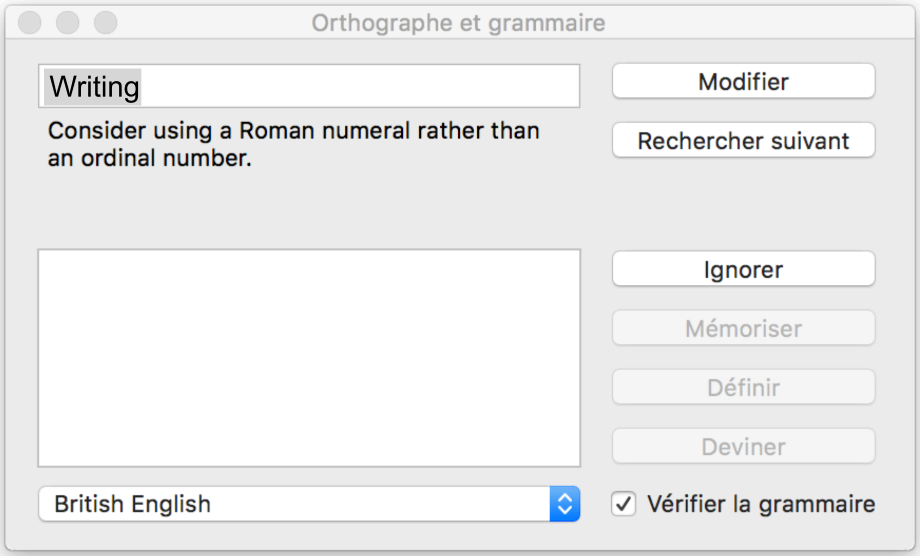
Spelling correction and edit distance

Spelling correction

- Two main application types:
  - Writign assistance

Spelling correction

- Two main application types:
  - Writign assistance



Spelling correction

- Two main application types:
  - Writing assistance
  - Correction of texts before NLP processing
- Two subtasks:
  - **Spelling error detection**
    - Maybe the hardest subtask, because many erroneous spellings produce existing words!
  - **Spelling error correction**
    - Autocorrect: *hte -> the*
    - Suggested correction
    - Suggested list of possible corrections (sorted)

Types and rates

- **Types**:
  - Non-word errors (easy to detect with a large lexicon) vs. real-word errors (harder to detect)
  - Typographical errors vs. “Cognitive” errors
  - Lexical errors vs. Grammatical errors
- **Rates**:
  - 26%: Web queries (Wang et al. 2003)
  - 13%: Retyping, no backspace (Whitelaw et al. on English & German)
  - 7%: Words corrected retyping on phone-sized “organiser”
  - 2%: Words uncorrected on “organiser” (Soukoreff and MacKenzie 2003)
  - 1-2%: Retyping (Kane and Wobbrock 2007, Gruden et al. 1983)

A non-word example

`acress`

Edit distance

- Creating candidate corrections
- Damereau-Levenshtein distance = minimal distance between two strings, based on a closed inventory of possible operations:
  - insertion of a character
  - deletion of a character
  - substitution of a character with another one
  - swap (transposition) between two adjacent characters
- Computed using dynamic programming
  - We fill a matrix whose (i,j) element stores the minimum number of operations needed to produce the j-prefix $w'_0 \ldots w'_j$ of the target word from the i-prefix $w_0 \ldots w_i$ of the source word

Levenshtein distance

$\textsc{LevenshteinDistance}(s_1, s_2)$

for $i \leftarrow 0$ **to** $|s_1|$
do $m[i, 0] = i$
for $j \leftarrow 0$ **to** $|s_2|$
do $m[0, j] = j$
for $i \leftarrow 1$ **to** $|s_1|$
do for $j \leftarrow 1$ **to** $|s_2|$
    do if $s_1[i] = s_2[j]$
        then $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]\}$
        else $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]+1\}$
return $m[|s_1|, |s_2|]$

Operations: insert (cost 1), delete (cost 1), replace (cost 1)

On our non-word example

- Words at a distance 1 from acress

Error	Candidate Correction	Correct Letter	Error Letter	Type
acress	actress	t	–	deletion
acress	cress	–	a	insertion
acress	caress	ca	ac	transposition
acress	access	c	r	substitution
acress	across	o	e	substitution
acress	acres	–	s	insertion
acress	acres	–	s	insertion

Edit distance and spelling errors

- 80% of errors are within edit distance 1
  - Almost all errors within edit distance 2
- We must also allow insertion of space or hyphen
  - *thisidea* -> *this idea*
  - *inlaw* -> *in-law*

anagement maagement maanagement
maangement magagement magement
mamagement mamangement manaagement manaement
managaement manageement manageemnt managegment
managemaent managemant managememt managemen managemenet
managementt managemet managemetn managemnent managemnet
managemnt managemrnt managemt managenent managenment managent
managerment managhement managmeent managmement managment managnment
manament manamgement mananement manangment manasgement
manegement manegment mangaement mangagement mangagment
mangament mangement manggement mangment
mangmt menagement mgmt mgnt
mnagement mngmnt mngmt

Candidate generation

- Several options:
  1. Run through dictionary, check edit distance with each word
  2. Generate all words within edit distance ≤ k (e.g. k = 1 or 2) and then intersect them with dictionary
  3. Use a character k-gram index and find dictionary words that share "most" k-grams with word
  4. Compute them fast with a Levenshtein finite state transducer
  5. Have a precomputed hash of words to possible corrections

Refinement: weighted operations

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

X \ Y (correct)	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	7	1	342	0	0	2	118	0	1	0	0	3	76	0	0	1	35	9	9	0	1	0	5	0
b	0	0	9	9	2	2	3	1	0	0	0	5	11	5	0	10	0	0	2	1	0	0	8	0	0	0
c	6	5	0	16	0	9	5	0	0	0	1	0	7	9	1	10	2	5	39	40	1	3	7	1	1	0
d	1	10	13	0	12	0	5	5	0	0	2	3	7	3	0	1	0	43	30	22	0	0	4	0	2	0
e	388	0	3	11	0	2	2	0	89	0	0	3	0	5	93	0	0	14	12	6	15	0	1	0	18	0
f	0	15	0	3	1	0	5	2	0	0	0	3	4	1	0	0	0	6	4	12	0	0	2	0	0	0
g	4	1	11	11	9	2	0	0	0	1	1	3	0	0	2	1	3	5	13	21	0	0	1	0	3	0
h	1	8	0	3	0	0	0	0	0	0	2	0	12	14	2	3	0	3	1	11	0	0	2	0	0	0
i	103	0	0	0	146	0	1	0	0	0	0	6	0	0	49	0	0	0	2	1	47	0	2	1	15	0
j	0	1	1	9	0	0	1	0	0	0	0	2	1	0	0	0	0	0	5	0	0	0	0	0	0	0
k	1	2	8	4	1	1	2	5	0	0	0	0	5	0	2	0	0	0	6	0	0	0	4	0	0	3
l	2	10	1	4	0	4	5	6	13	0	1	0	0	14	2	5	0	11	10	2	0	0	0	0	0	0
m	1	3	7	8	0	2	0	6	0	0	4	4	0	180	0	6	0	0	9	15	13	3	2	2	3	0
n	2	7	6	5	3	0	1	19	1	0	4	35	78	0	0	7	0	28	5	7	0	0	1	2	0	2
o	91	1	1	3	116	0	0	0	25	0	2	0	0	0	0	14	0	2	4	14	39	0	0	0	18	0
p	0	11	1	2	0	6	5	0	2	9	0	2	7	6	15	0	0	1	3	6	0	4	1	0	0	0
q	0	0	1	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	14	0	30	12	2	2	8	2	0	5	8	4	20	1	14	0	0	12	22	4	0	0	1	0	0
s	11	8	27	33	35	4	0	1	0	1	0	27	0	6	1	7	0	14	0	15	0	0	5	3	20	1
t	3	4	9	42	7	5	19	5	0	1	0	14	9	5	5	6	0	11	37	0	0	2	19	0	7	6
u	20	0	0	0	44	0	0	0	64	0	0	0	0	2	43	0	0	4	0	0	0	0	2	0	8	0
v	0	0	7	0	0	3	0	0	0	0	0	1	0	0	1	0	0	0	8	3	0	0	0	0	0	0
w	2	2	1	0	1	0	0	2	0	0	1	0	0	0	0	7	0	6	3	3	1	0	0	0	0	0
x	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0
y	0	0	2	0	15	0	1	7	15	0	0	0	2	0	6	1	0	7	36	8	5	0	0	1	0	0
z	0	0	0	7	0	0	0	0	0	0	0	7	5	0	0	0	0	2	21	3	0	0	0	0	3	0

(Kernighan, Church, Gale 1990)

Refinement: weighted operations

- Such data makes it possible to compute substitution probabilities
- Can be generalised to all 4 types of operations
  - Use add-1 smoothing to allow for unseen operations

> **Channel model**

Candidate Correction	Correct Letter	Error Letter	x|w	P(x|word)
actress	t	-	c|ct	.000117
cress	-	a	a|#	.00000144
caress	ca	ac	ac|ca	.00000164
access	c	r	r|c	.000000209
across	o	e	e|o	.0000093
acres	-	s	es|e	.0000321
acres	-	s	ss|s	.0000342

Language model

- Noisy channel approach: multiply channel model by language model
- We want to prefer more frequent output tokens
  - Simplest idea: unigram probability

Candidate Correction	Correct Letter	Error Letter	x|w	P(x|word)	P(word)	10^9 *P(x|w)P(w)
actress	t	-	c|ct	.000117	.0000231	2.7
cress	-	a	a|#	.00000144	.000000544	.00078
caress	ca	ac	ac|ca	.00000164	.00000170	.0028
access	c	r	r|c	.000000209	.0000916	19
across	o	e	e|o	.0000093	.000299	2.8
acres	-	s	es|e	.0000321	.0000318	1.0
acres	-	s	ss|s	.0000342	.0000318	1.0

Language model

- Noisy channel approach: multiply channel model by language model
- We want to prefer more frequent output tokens
  - Simplest idea: unigram probability

Candidate Correction	Correct Letter	Error Letter	x|w	P(x|word)	P(word)	10^9 *P(x|w)P(w)
actress	t	-	c|ct	.000117	.0000231	2.7
cress	-	a	a|#	.00000144	.000000544	.00078
caress	ca	ac	ac|ca	.00000164	.00000170	.0028
access	c	r	r|c	.000000209	.0000916	19
across	**o**	**e**	**e|o**	**.0000093**	**.000299**	**2.8**
acres	-	s	es|e	.0000321	.0000318	1.0
acres	-	s	ss|s	.0000342	.0000318	1.0

Language model

- Better: **incorporating context**
  - Simplest way: bigram language model
  - Better: with linear interpolation with unigram model to avoid unseen bigrams
    $P_{li}(w_k | w_{k-1}) = \lambda.P_{uni}(w_k) + (1-\lambda).P_{mle}(w_k | w_{k-1})$
    (where $P_{mle}(w_k | w_{k-1}) = \#(w_k | w_{k-1})/\#w_{k-1}$) is called the maximum likelihood estimate)
  - The model can also be smoothed (e.g. add-1)
- Imagine the following context for our example:
  a stellar and versatile acress whose combination of sass and glamour…
- Counts from the Corpus of Contemporary American English with add-1 smoothing
  - P(actress|versatile)=.000021 P(whose|actress) = 0.0010
    P(across|versatile) =.000021 P(whose|across) = 0.000006
    P("versatile actress whose") = .000021 x .0010 = 210.10^{-10}
    P("versatile across whose") = .000021 x .000006 = 1.10^{-10}

Language model

- Better: **incorporating context**
  - Simplest way: bigram language model
  - Better: with linear interpolation with unigram model to avoid unseen bigrams
    $P_{li}(w_k | w_{k-1}) = \lambda.P_{uni}(w_k) + (1-\lambda).P_{mle}(w_k | w_{k-1})$
    (where $P_{mle}(w_k | w_{k-1}) = \#(w_k | w_{k-1})/\#w_{k-1}$) is called the maximum likelihood estimate)
  - The model can also be smoothed (e.g. add-1)
- Imagine the following context for our example:
  a stellar and versatile acress whose combination of sass and glamour…
- Counts from the Corpus of Contemporary American English with add-1 smoothing
  - P(actress|versatile)=.000021 P(whose|actress) = 0.0010
    P(across|versatile) =.000021 P(whose|across) = 0.000006
    P("versatile actress whose") = .000021 x .0010 = 210.10^{-10}
    P("versatile across whose") = .000021 x .000006 = 1.10^{-10}
  - The best correction is "actress"!

Real-word errors

- Much harder to detect
- Are they frequent?
  - 25-40% of spelling errors are real words (Kukich 1992)
  - More real-word errors in content produced on smartphones since the generalisation of semi-automatic correction
- General idea:
  - Guess which words *could* be errors
  - Produce candidates but **include the original word amongst them**
    - **Homophones** are important (phonetisation)
  - Choose best candidate with noisy channel model

The CYK algorithm

Classical CFG parsing algorithms

- Programming languages are defined by CFGs that are not ambiguous
  - Algorithms for non-ambiguous (“deterministic”) CFGs are relatively simple (LR, LALR…)
- Natural language is ambiguous. This resulted in the development of algorithms for parsing general (i.e. not only deterministic) CFGs
  - **CYK**
  - Earley
  - GLR

A membership problem

- The basic Cocke–Younger–Kasami, a.k.a. CYK (sometimes CKY) algorithm, solves a **membership problem**: it is not a parsing algorithm *per se*
  - An implementation of the CYK algorithm is a **recogniser**, not a parser
  - But it can easily be adapted to become a parsing algorithm
- The membership problem is simple: given an CFG and an input string, answer the following question:
  Is the input string in the language defined by the CFG?
- The CFG must be in **Chomsky normal form**, i.e. all rewriting rules are of one of the following forms:
  - $S \longrightarrow A\ B$
  - $S \longrightarrow a$
  - $S \longrightarrow \varepsilon$
- It is a bottom-up algorithm based on dynamic programming

Basic idea

- Iterative algorithm
- Let $u = x_1 x_2 \ldots x_n$ be a string to be tested for membership
  - Step 1: For each substring of u of length 1, find the set $\mathcal{S}_{i\text{-}1,i}$ of non-terminals A with a rule A $\longrightarrow x_i$
  - Step 2: for each substring $x_i x_{i+1}$ of u of length 2 find the set $\mathcal{S}_{i\text{-}1,i+1}$ of variables A that derives A $\longrightarrow x_i x_{i+1}$
  - …
  - Step n: for the whole string $u = x_1 x_2 \ldots x_n$, find the set $\mathcal{S}_{0,n}$ of variables A that derives A $\longrightarrow x_1 x_2 \ldots x_n$
  - Iff $\mathcal{S}_{0,n}$ contains the axiom S, then u is in the language defined by the grammar.

Diagonal table

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
w_1	w_2	w_3	w_4

CYK on an example

$S \rightarrow AB$

$A \rightarrow CC \mid a \mid c$

$B \rightarrow BC \mid b$

$C \rightarrow CB \mid BA \mid c$

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
{S,C}	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
{S,C}	$\varnothing$	$\mathcal{S}_{2,4}$	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
{S,C}	∅	{C}	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

$S \rightarrow AB$

$A \rightarrow CC \mid a \mid c$

$B \rightarrow BC \mid b$

$C \rightarrow CB \mid BA \mid c$

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{2,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

$\mathcal{S}_{0,4}$			
{C}	$\mathcal{S}_{1,4}$		
{S,C}	$\varnothing$	{C}	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

$\mathcal{S}_{0,4}$			
{C}	{B}		
{S,C}	∅	{C}	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

{S,A,C}			
{C}	{B}		
{S,C}	$\varnothing$	{C}	
{A,C}	{B}	{B}	{A}
c	b	b	a

CYK on an example

S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

$\mathcal{S}_{0,4}$			
$\mathcal{S}_{0,3}$	$\mathcal{S}_{1,4}$		
$\mathcal{S}_{0,2}$	$\mathcal{S}_{1,3}$	$\mathcal{S}_{2,4}$	
$\mathcal{S}_{0,1}$	$\mathcal{S}_{1,2}$	$\mathcal{S}_{2,3}$	$\mathcal{S}_{3,4}$
c	b	b	a

{**S**,A,C}			
{C}	{B}		
{S,C}	∅	{C}	
{A,C}	{B}	{B}	{A}
c	b	b	a

=> the string is in the language defined by the CFG

Converting a CFG into Chomsky normal form (aka binarisation)

- We must eliminate:
  - Rules where the start symbol (the axiom) is on the right hand side
  - Rules with non-solitary terminal symbols
  - Rules with right-hand sides with more than 2 non-terminal symbols
  - Unit rules (rules with only 1 non-terminal symbol on the right-hand side)
  - Epsilon-rules (rules with an empty right-hand side)

Converting a CFG into Chomsky normal form (aka binarisation)

- **Eliminate the start symbol from right-hand sides**
  - Introduce a new start symbol S0, and a new rule
    $S_0 \rightarrow S$
    where S is the previous start symbol
- **Eliminate rules with non-solitary terminals**
  - To eliminate each rule
    $A \rightarrow X_1 \ldots a \ldots X_n$
    with a terminal symbol a being not the only symbol on the right-hand side, introduce, for every such terminal, a new non-terminal symbol N_a, and a new rule
    $N_a \rightarrow a$
  - Change every rule
    $A \rightarrow X_1 \ldots a \ldots X_n$
    to
    $A \rightarrow X_1 \ldots N_a \ldots X_n$
  - If several terminal symbols occur on the right-hand side, simultaneously replace each of them by its associated non-terminal symbol.

Converting a CFG into Chomsky normal form (aka binarisation)

- **Eliminate right-hand sides with more than 2 non-terminals**
  - Replace each rule
    $A \rightarrow X_1\ X_2\ ...\ X_n$
    with more than 2 nonterminals $X_1,...,X_n$ by rules
    $A \rightarrow X_1\ A_1$,
    $A_1 \rightarrow X_2\ A_2$,
    ... ,
    $A_{n-2} \rightarrow X_{n-1}\ X_n$,
    where A_i are new non-terminal symbols.
- **Eliminate unit rules**
  - A unit rule is a rule of the form
    $A \rightarrow B$,
    where A, B are non-terminal symbols. To remove it, for each rule
    $B \rightarrow X_1\ ...\ X_n$,
    where $X_1\ ...\ X_n$ is a string of non-terminals and terminals, add rule
    $A \rightarrow X_1\ ...\ X_n$
    unless this is a unit rule which has already been (or is being) removed.

Converting a CFG into Chomsky normal form (aka binarisation)

- For more details, simply go to Wikipedia (on which both previous slides are based):

  https://en.wikipedia.org/wiki/Chomsky_normal_form

NLP practical assignment (TD2)

Goal and instructions

- **Develop a basic probabilistic parser for French that is based on the CYK algorithm and the PCFG model and that is robust to unknown words**

- You will extract a PCFG from the training corpus provided, made of:
    - a probabilistic context-free grammar whose terminals are part-of-speech tags
    - a probabilistic lexicon, i.e. triples of the form (token, part-of-speech tag, probability) such that the sum of the probabilities for all triples for a given token sums to 1.
- You will reimplement the CYK algorithm, adapting it a bit in two ways: (1) to handle the existence of a lexicon; (2) to only retain the most probable way to rewrite an instantiated symbol, in order to directly extract the best (most probable) parse tree for each sentence
- Use the SEQUOIA treebank v6.0 (file in the GitHub, bracketed format):
    - Split it into 3 parts (80% / 10% / 10%)
    - Use the 80% for training (extract CFG rules + learn CFG rule probabilities)
    - Use the first 10% for development purposes (whatever you want to use it for)
    - Use the last 10% for evaluating your parser
    - IMPORTANT: Ignore the functional labels: whenever you find a hyphen in a non-terminal name, ignore it and everything that follows
      E.g.: ((SENT (PP-MOD (P En) (NP (NC 1996))) (PONCT ,) (NP-SUJ (DET la) (NC municipalité)) (VN (V étudie)) (NP-OBJ (DET la) (NC possibilité) (PP (P d') (NP (DET une) (NC construction) (AP (ADJ neuve))))) (PONCT .)))

Goal and instructions

- **Develop a basic probabilistic parser for French that is based on the CYK algorithm and the PCFG model and that is robust to unknown words**
- Your OOV module will assign a (unique) part-of-speech to any token not included in the lexicon extracted from the training corpus.
  - The underlying idea is to assign to an OOV the part-of-speech of a "similar" word.
  - This similarity will be computed as a combination of **formal similarity** (to handle spelling errors) and **embedding similarity** (as measured by cosine similarity, i.e. scalar product between normalised vectors), to handle both spelling errors and genuine unknown words
  - You must design a reasonable way to combine these two similarities.
  - For embedding similarity, you will use the Polyglot embedding lexicon for French https://sites.google.com/site/rmyeid/projects/polyglot
    - See the tutorial with code snippets here: https://nbviewer.jupyter.org/gist/aboSamoor/6046170

Assignment deliverable

- An archive containing your code and a short report
  - See details on the GitHub page of the assignment
- Please respect the instructions given in the assignment description on the GitHub
  - Failure to do so will be penalised

https://github.com/edupoux/MVA_2019_SL/tree/master/TD_%232

That’s all for today!