Apunts del Taller de Nous Usos de la Informàtica

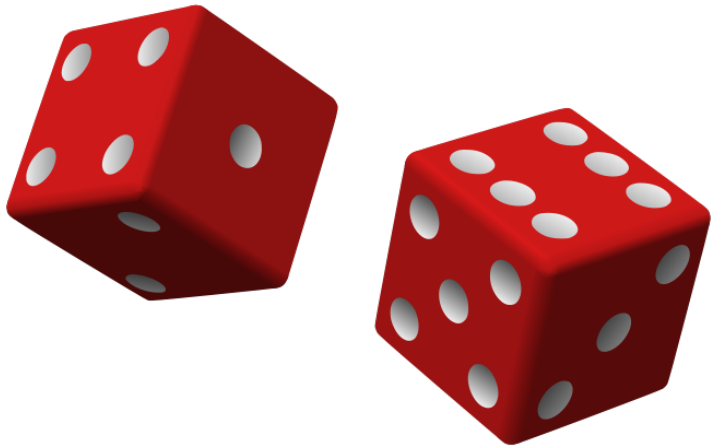
Jordi Vitrià

Universitat de Barcelona

10 de setembre de 2019

UNIVERSITAT DE BARCELONA

Lliçó: Models Probabilístics i Llenguatge Natural



Eines pel Processament del Llenguatge Natural

- Què podem fer amb un *corpus* de milions de paraules de llenguatge natural?
- En aquesta lliçó veurem dos exemples, basats en la construcció d'un **model probabilístic**, que corresponen a diferents eines que són útils en el context de la interacció per escrit amb un usuari:
  - Fer un segmentador de frases, o com decidir que la frase `'wheninthecourseofhumaneventsitbecomesnecessary'`, que té aproximadament 35.000.000.000.000 maneres possibles de segmentar-se, ha de partir-se d'una manera concreta.
  - Fer un corrector de paraules (*spelling correction*) o com decidir que l'usuari, tot i haver escrit `acomodation` volia escriure `accomodation`.

El Corpus

- L'any 2006, Google va fer públic (`http://tinyurl.com/ngrams`) un conjunt de dades estadístiques extretes d'una sèrie de documents en llenguatge natural que contenien més de 1.000.000.000.000 paraules en anglès.
  - Number of tokens: 1,024,908,267,229
  - Number of sentences: 95,119,665,584
  - Number of unigrams: 13,588,391
  - Number of bigrams: 314,843,401
  - Number of trigrams: 977,069,902
  - Number of fourgrams: 1,313,818,354
  - Number of fivegrams: 1,176,470,663

El Corpus

- Les dades resumeixen els documents **contant** el nombre de vagades que hi apareix una determinada paraula i també les seqüències de dues, tres, quatre i cinc paraules concretes.
  - Per exemple, `the` apareix 23.000.000.000 vegades (és un 2.2% del total de paraules) i és la paraula més freqüent.
  - La paraula `rebating` apareix 12.750 vegades (és un 0.000001% del total de paraules), les mateixes vegades que `fnuny` (aparentment, un error de la paraula `funny`).
  - Respecte a les seqüències de tres paraules, `Find all posts` apareix 13 milions de vegades (0.001%), tant sovint com `each of the`, però molt menys que les 100 milions de vegades de `All Rights Reserved` (0.01%).

Definicions I

- Una col·lecció de textos s'anomena **corpus**.
- El corpus està format per una seqüència de **tokens**, paraules i signes de puntuació/blancs.
- Cada token diferent s'anomena **tipus**, i per tant la frase `Run, Lola Run` té 4 tokens (la coma conta) però només 3 tipus.
- El conjunt de tots els tipus s'anomena **vocabulari**. El corpus de Google té un vocabulari de 13.000.000 de tipus. La majoria dels tipus tenen una freqüència petita: els 10 tipus més freqüents cobreixen casi una tercera part dels tokens, els 1000 més freqüents dues terceres parts, i els 100.000 més freqüents cobreixen el 98% dels tokens.
- Una seqüència d'un token és un **unigrama**, una de dos token és un **bigrama** i una de n token és un **ngrama**.

Definicions II

- Direm que un cert tipus té una probabilitat P, com per exemple $P(\texttt{the}) = 0.022$, quan la freqüència del tipus al corpus sigui d'un $P \times 100$ %.
- Si W és una seqüència de tokens, W_3 representa el tercer token i $W_{1:3}$ representa la seqüència del primer al tercer.
- $P(W_i = \texttt{the} | W_{i-1} = \texttt{of})$ s'anomena la probabilitat condicional de `the`, donat que `of` és el token anterior.

Segmentació

- Hi ha ocasions en que ens poden arribar seqüències de paraules sense separacions en les que cal decidir com separar-ho. Per exemple, els noms dels lloc web (URLs): `www.lordoftherings.com`. Aquest problema s'anomena **segmentació de paraules**.
- Com podem corregir ortogràficament un text en el que s'ha fet un error d'ajuntar paraules? Primer hem de segmentar!
- Considerem la seqüència `insufficientnumbers`. Si consultem el corpus, veurem que `insufficient numbers` hi apareix 20751 vegades i `in sufficient numbers` hi apareix 32378 vegades. Tot i que la segona opció sembla més probable, no tenim seguretat. Cal una metodologia que ens permeti agregar totes les evidències i combinar-les de forma adequada.

Models Probabilistics

- Hi ha una metodologia adequada a la resolució de problemes relacionats amb el manejament de la incertesa:
  1. Definim un **model probabilistic**. En el nostre cas, l'anomenarem **model de llenguatge** i serà una distribució de probabilitats sobre tots els possibles *strings* que es puguin formar a partir dels tipus definits. Els paràmetres d'aquest models s'aprendran del corpus i ens serviran per assignar una probabilitat a cada possible solució.
  2. **Enumerem les solucions candidates**. En el nostre cas, hem d'enumerar de forma eficient totes les possibles segmentacions de la nostra seqüència.
  3. **Escollim la solució amb màxima probabilitat**. En el nostre cas, aplicarem el model de llenguatge a cada possible solució candidata.

- En terminologia probabilística, el que volem s'escriu així:

$$best = argmax_{c \in candidates} P(c) \tag{1}$$

Comencem per definir el **model probabilístic de llenguatge**.

- Teoricament, la probabilitat d'una seqüència de paraules és la probabilitat de cada una de les paraules, donat el context de cada una (que es pot definir com les paraules que la precedeixen a la seqüència):

$$P(W_{1:n}) = \prod_{k=1,\ldots,n} P(W_k | W_{1:k-1}) \tag{2}$$

Model probabilístic de llenguatge

- És evident que no hi ha prou dades per calcular aquestes probabilitats, atès que hauríem de calcular $P(W_k|W_{1:k-1})$ per cada possible seqüència $W_{1:k-1}$.
- Podem considerar el model simplificat que només pren en consideració els unigrames:

$$P(W_{1:n}) \approx \prod_{k=1,\ldots,n} P(W_k) \tag{3}$$

- En aquest cas, per segmentar `lacasade`, hem de considerar entre els candidats el conjunt $\{\texttt{la}, \texttt{casa}, \texttt{de}\}$ i calcular la seva probabilitat $P(\texttt{la}) \times P(\texttt{casa}) \times P(\texttt{de})$. Si cap altre conjunt un producte més gran de probabilitats, llavors ja hem trobat la solució.

Com seleccionem els candidats?

- Un string de n caràcters té 2^{n-1} segmentacions possibles, atès que hi ha $n-1$ llocs possibles entre lletra i lletra.
- Per tant, l'enumeració de tots els candidats possibles no és una opció i hem de buscar una opció més eficient.
- Podem implementar una aproximació *recursiva* definida a partir de segmentació entre una primera paraula (la longitud de la qual està limitada per la longitud de la paraula més llarga del corpus) i la resta del text.
- De tots els possibles candidats, la que té $P(first) \times P(remaining)$ màxima és la millor.

Com seleccionem els candidats?

- Suposem que tenim `wheninthecourseofhumaneventsitbecomesnecessary`.
  L'algorisme determinaria que `when` és la primera paraula d'aquesta manera:

first	P(*first*)	P(*remaining*)	P(*first*) × P(*remaining*)
w	$2 \cdot 10^{-4}$	$2 \cdot 10^{-33}$	$6 \cdot 10^{-37}$
wh	$5 \cdot 10^{-6}$	$6 \cdot 10^{-33}$	$3 \cdot 10^{-38}$
whe	$3 \cdot 10^{-7}$	$3 \cdot 10^{-32}$	$7 \cdot 10^{-39}$
when	$6 \cdot 10^{-4}$	$7 \cdot 10^{-29}$	$4 \cdot 10^{-32}$
wheni	$1 \cdot 10^{-16}$	$3 \cdot 10^{-30}$	$3 \cdot 10^{-46}$
whenin	$1 \cdot 10^{-17}$	$8 \cdot 10^{-27}$	$8 \cdot 10^{-44}$

El programa

```
@memo
def segment(text):
    "Return a list of words that is the best segmentation of text."
    if not text: return []
    candidates = ([first]+segment(rem) for first,rem in splits(text))
    return max(candidates, key=Pwords)

def splits(text, L=20):
    "Return a list of all possible (first, rem) pairs, len(first)<=L."
    return [(text[:i+1], text[i+1:])
            for i in range(min(len(text), L))]

def Pwords(words):
    "The Naive Bayes probability of a sequence of words."
    return product(Pw(w) for w in words)
```

El programa

```
@memo
def segment(text):
    "Return a list of words that is the best segmentation of text."
    if not text: return []
    candidates = ([first]+segment(rem) for first,rem in splits(text))
    return max(candidates, key=Pwords)

def splits(text, L=20):
    "Return a list of all possible (first, rem) pairs, len(first)<=L."
    return [(text[:i+1], text[i+1:])
            for i in range(min(len(text), L))]

def Pwords(words):
    "The Naive Bayes probability of a sequence of words."
    return product(Pw(w) for w in words)
```

- `product` és una funció que multiplica una llista de nombres.
- `Pw` és una funció que retorna la probabilitat d'una paraula de la llista dels unigrames.

demo és un *decorador* de Python que manté en memòria els resultats de les crides anteriors d'una funció de manera que no s'han de recalcular si es tornen a necessitar.

```
def memo(f):
    "Memoize function f."
    table = {}
    def fmemo(*args):
        if args not in table:
            table[args] = f(*args)
        return table[args]
    fmemo.memo = table
    return fmemo
```

```
@memo
def fib(n):
    if n==1:
        return 1
    if n==0:
        return 0
    return fib(n-1) + fib(n-2)

def fibonacci(n):
    import time
    import math
    t1 = time.clock()
    a = fib(n)
    t2 = time.clock()
    print "El temps de proces ha estat %1.3f ms" % ((t2-t1)*1000)

>>> fibonacci(300)
El temps de proces ha estat 0.845 ms
>>> fibonacci(301)
El temps de proces ha estat 0.006 ms
>>>
```

```
>>> splits("nousususos")
[('n', 'ousususos'), ('no', 'usususos'), ('nou', 'susos'),
('nous','usos'), ('nousu', 'sos'), ('nousus', 'os'),
('noususo','s'), ('nousususos', '')]
>>> segment("nousususos")
['nous', 'usos']
>>>
```

El programa

- Sense `memo` la crida a la funció `segment` amb un string de n caracters implica 2^n crides recursives. Amb `memo` ho transformem en un procés de *programació dinàmica* i només en fa n. Cada crida condidera $O(L)$ segmentacions (on L és la longitud màxima d'una paraula), i per cada una evalua $O(n)$ probabilitats, per tant la complexitat és $O(n^2 L)$.
- Respecte `Pw`, accedeix al fitxer amb els estadístics (`count`) de cada paraula i estima la probabilitat com `count(word)/N`, on N és el nombre d'elements del corpus.

Com calculem Pw

- Què fem si la paraula no és al diccionari? (no podem retornar 0)!
- La paraula menys vista del diccionari té una probabilitat de $12710/N$ i per tant una paraula no vista hauria de tenir una probabilitat menor.
- Una heurística que funciona és assignar una probabilitat inicial de $12710/N$, però decrementem per un factor 10 per cada lletra de la paraula no vista (com més llarga, més improbable).

Primers resultats

```
>>> segment('itisatruthuniversallyacknowledged')
['it', 'is', 'a', 'truth', 'universally', 'acknowledged']
>>> segment('itwasabrightcolddayinaprilandtheclockswerestrikingthirteen')
['it', 'was', 'a', 'bright', 'cold', 'day', 'in', 'april', 'and', 'the', 'clocks',
'were', 'striking', 'thirteen']
>>> segment('itwasthebestoftimesitwastheworstoftimesitwastheageofwisdomitwastheage
offoolishness')
['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times',
'it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'of',
'foolishness']
>>> segment('asgregorsamsaawokeonemorningfromuneasydreamshefoundhimselftransformed
inhisbedintoagiganticinsect')
['as', 'gregor', 'samsa', 'awoke', 'one', 'morning', 'from', 'uneasy', 'dreams', 'he',
'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'gigantic',
'insect']
>>> segment('inaholeinthegroundtherelivedahobbitnotanastydirtywetholefilledwiththe
endsofwormsandanoozysmellnoryetadrybaresandyholewithnothinginittositdownonortoeat
itwasahobbitholeandthatmeanscomfort')
['in', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a', 'hobbit', 'not', 'a',
'nasty', 'dirty', 'wet', 'hole', 'filled', 'with', 'the', 'ends', 'of', 'worms', 'and',
'an', 'oozy', 'smell', 'nor', 'yet', 'a', 'dry', 'bare', 'sandy', 'hole', 'with',
'nothing', 'in', 'it', 'to', 'sitdown', 'on', 'or', 'to', 'eat', 'it', 'was', 'a',
'hobbit', 'hole', 'and', 'that', 'means', 'comfort']
```

Anàlisi dels resultats

- Un dels errors que hi ha és que no ha segmentat les paraules `sitdown`.
- El motiu és que el producte de les probabilitats de les dues paraules (que tenen probabilitats 0.003% i 0.04% respectivament) és lleugerament inferior que la de `sitdown`. Però si miréssim les probabilitats dels bigrames, veuriem que la probabilitat de `sit down` és 100 vegades superior a la de `sitdown`!
- Per tant, aquest error es pot resoldre considerant un model de bigrames:

$$P(W_{1:n}) = \prod_{k=1,\ldots,n} P(W_k|W_{k-1}) \tag{4}$$

Models de bigrames

- La taula amb els valors de tots els bigrames no cap a la memòria!
- Si considerem els bigrames que apareixen més de 100.000 vegades obtenim una taula de 250.000 entrades, que si que hi cap. Llavors podem estimar $P(\texttt{down}|\texttt{sit})$ com `count(sit down)/count(sit)`.
- Si un bigrama no apareix a la taula, llavors usem l'estimació dels unigrames.

```
def cPw(word, prev):
    "The conditional probability P(word | previous-word)."
    try:
        return P2w[prev + ' ' + word]/float(Pw[prev])
    except KeyError:
        return Pw(word)

P2w = Pdist(datafile('count2w'), N)
```

Models de bigrames

- La funció `cPw` no genera una distribució de probabilitats (pot sumar més que 1), tot i que a la pràctica funciona bé. Per aquest motiu s'anomena *stupid backoff*.
- Els nous resultats són que ara trobem `sit down` 1.698 vegades més probable que `sitdown` perquè `sit down` té una freqüència alta i `to sitdown` no:

```
>>> cPw('sit', 'to')*cPw('down', 'sit') / cPw('sitdown', 'to')
1698.0002330199263
```

Models de bigrames

- Abans de generar una versió del nostre model de segmentació podem arreglar dos petits problemes addicionals:
  1. Quan `segment` afegeix una nova paraula a la seqüència de n paraules segmentades en el *remainder*, crida a la funció `Pwords` per multiplicar les $n+1$ probabilitats, però de fet ja les havia multiplicat! Seria més eficient recordar la probabilitat del *remainder* i afegir-hi només una multiplicació.
  2. Quan apliquem la funció `Pwords` a una seqüència llarga de paraules podem tenir problemes *d'underflow*. El nombre més petit que podem representar és 4.9×10^{-324} i qualsevol cosa més petita és 0. Una solució senzilla és fer una suma de logaritmes enlloc d'una multiplicació.

Models de bigrames

- La funció `segment2` fa $O(nL)$ crides recursives i cada una considera $O(L)$ particions, de manera que l'algorisme sencer té una complexitat $O(nL^2)$!

```
from math import log10

@memo
def segment2(text, prev='<S>'):
    "Return (log P(words), words), where words is the best segmentation."
    if not text: return 0.0, []
    candidates = [combine(log10(cPw(first, prev)), first, segment2(rem, first))
                  for first,rem in splits(text)]
    return max(candidates)

def combine(Pfirst, first, (Prem, rem)):
    "Combine first and rem results into one (probability, words) pair."
    return Pfirst+Prem, [first]+rem
```

- L'algorisme que hem desenvolupat és una implementació de l'**algorisme de Viterbi**.

Un corrector ortogràfic

- El nostre objectiu és determinar quina paraula c és la més probable donada una paraula introduïda amb un teclat, amb possibles errors ortogràfics i/o tipogràfics, `w`.
- Seguirem el models estàndard: buscar la c que maximitza $P(c|w)$.
- Per fer-ho, cal tenir en compte les característiques del problema: si tenim `w = 'thew'`, la paraula més probable podria ser `the` (suposant que hem afegit una lletra al final), però hi ha altres paraules candidates: `thaw` (un canvi de vocal), `thew` (aquesta paraula existeix!), `threw`, etc.
- El problema ens obliga a combinar dos criteris: la probabilitat de la paraula i la probabilitat d'haver fer un cert error ortogràfic i/o tipogràfic.

El teorema de Bayes

- Quan hem de **combinar diverses fonts incertes**, la millor opció és fer servir la *fòrmula de Bayes*:

$$P(c|w) = P(w|c)P(c)/P(w) \tag{5}$$

que compleix,

$$argmax_c P(c|w) = argmax_c P(w|c)P(c) \tag{6}$$

- $P(c)$, la probabilitat a priori de que c sigui la paraula correcta, s'anomena el **model de llenguage**.
- $P(w|c)$, la probabilitat que l'usuari hagui escrit w quan volia escriure c, s'anomena el **model de l'error**.

El corpus

- Per aprendre el model de l'error necessitem un corpus, però el que hem vist abans no serveix directament.
- La nostra opció serà generar un corpus de correccions de forma sintètica, considerant els errors d'edició que es poden fer quan escribim una paraula:
  1. Podem *borrar* una lletra: `thew` $\rightarrow$ `the`.
  2. Podem *insertar* una lletra: `the` $\rightarrow$ `thwe`.
  3. Podem *reemplaçar* una lletra: `the` $\rightarrow$ `tha`.
  4. Podem *transposar* dues lletres adjacents: `the` $\rightarrow$ `teh`.
- Totes aquestes edicions estan a distància 1 de la paraula adicional. Si per corregir-la n'hen de fer dues, estarem a distància 2, etc.

El corpus

- Per estimar les probabilitats d'una certa edició (p.e. canviar una e per una r) podem fer servir repositoris públics d'errors tipogràfics, com per exemple `http://www.dcs.bbk.ac.uk/ ROGER/corpora.html` on hi ha 40.000 exemples de paraules mal escrites.
- Amb això podem estimar probabilitats:

w	c	$w \mid c$	$P(w \mid c)$	$P(c)$	$10^9 P(w \mid c) P(c)$
thew	the	ew | e	0.000007	0.02	144.
thew	thew		0.95	0.00000009	90.
thew	thaw	e | a	0.001	0.0000007	0.7
thew	threw	h | hr	0.000008	0.000004	0.03
thew	thwe	ew | we	0.000003	0.00000004	0.0001

Edicions i probabilitats

- Per crear $P(w|c)$ hem de fer una funció, `Pedit`, que ens doni la probabilitat d'una edició simple a partir del corpus. Les edicions més complexes les podem estimar a partir de concatenacions d'edicions simples: `hallow` $\rightarrow$ `hello` = `a|e + ow|o`.
- Per completitud, ens cal definir la probabilitat de que l'usuari hagui escrit la paraula bé. Ho definirem de forma arbitrària, assumint que hi ha un error cada 20 paraules escrites (evidentment, això depèn de l'usuari!). Per tant, `Pedit('')=0.95`.

El programa

- Els candidats estan generats per `edits`, que rep una paraula i retorna un diccionari del tipus {`word:edit`} que ens indica les possibles correccions. Com implementem de forma eficient aquesta funció? (si considerem totes les edicions possibles de la paraula `acommodations` tenim 233.166 solucions possibles, de les que només 11 són al vacabulari!).
- L'estrategia que podem seguir és la següent: pre-calcularem tots els prefixes de totes les paraules del vocabulari i llavors cridarem recursivament la funció `editsR`, dividint la paraula en una cap i un cos i assegurant que el cap està sempre a la llista de prefixes.

El programa

```
def edits(word, d=2):
    "Return a dict of {correct: edit} pairs within d edits of word."
    results = {}
    def editsR(hd, tl, d, edits):
        def ed(L,R): return edits+[R+'|'+L]
        C = hd+tl
        if C in Pw:
            e = '+'.join(edits)
            if C not in results: results[C] = e
            else: results[C] = max(results[C], e, key=Pedit)
        if d <= 0: return
        extensions = [hd+c for c in alphabet if hd+c in PREFIXES]
        p = (hd[-1] if hd else '<') ## previous character
        ## Insertion
        for h in extensions:
            editsR(h, tl, d-1, ed(p+h[-1], p))
```

El programa

```
        if not tl: return
        ## Deletion
        editsR(hd, tl[1:], d-1, ed(p, p+tl[0]))
        for h in extensions:
            if h[-1] == tl[0]: ## Match
                editsR(h, tl[1:], d, edits)
            else: ## Replacement
                editsR(h, tl[1:], d-1, ed(h[-1], tl[0]))
        ## Transpose
        if len(tl)>=2 and tl[0]!=tl[1] and hd+tl[1] in PREFIXES:
            editsR(hd+tl[1], tl[0]+tl[2:], d-1,
                   ed(tl[1]+tl[0], tl[0:2]))
    ## Body of edits:
    editsR('', word, d, [])
    return results

PREFIXES = set(w[:i] for w in Pw for i in range(len(w) + 1))
```

El programa

- La funció `edits` es comporta així:

```
>>> edits('adiabatic', 2)
{'adiabatic': '', 'diabetic': '<a|<+a|e', 'diabatic': '<a|<'}
```

El programa

Hi ha dues funcions: `correct`, que ens retorna la correcció més probable i `corrections` que ho aplica a tot un text.

```
def corrections(text):
    "Spell-correct all words in text."
    return re.sub('[a-zA-Z]+', lambda m: correct(m.group(0)), text)

def correct(w):
    "Return the word that is the most likely spell correction of w."
    candidates = edits(w).items()
    c, edit = max(candidates, key=lambda (c,e): Pedit(e) * Pw(c))
    return c
```

```
def Pedit(edit):
    "The probability of an edit; can be '' or 'a|b' or 'a|b+c|d'."
    if edit == '': return (1. - p_spell_error)
    return p_spell_error*product(P1edit(e) for e in edit.split('+'))

p_spell_error = 1./20.

P1edit = Pdist(datafile('count1edit')) ## Probabilities of single edits
```

El programa

- El corrector funciona així:

```
>>> correct('vokabulary')
'vocabulary'

>>> correct('embracable')
'embraceable'

>>> corrections('thiss is a teyst of acommodations for korrections
of mispellings of particuler wurds.')
'this is a test of acommodations for corrections of mispellings
of particular words.'
```

- Algunes de les paraules que no ha corregit bé és a causa de que apareixen mal escrites al corpus original (`mispellings` apareix més de 18.000 vagades!) i tot i que la paraula original és més probable, no té prou 'força' per fer guanyar l'edició.

Conclusions

- L'esquema que hem vist implementat pot servir per moltes tasques anàlogues:
  - Fer un detector de correu spam (cada dia s'enviem més de 100.000.000.000 de correus spam) basat en n-grames de paraules (`our country is Nigeria`) i en n-grames de caràcters (`vIagra`).
  - Fer un classificador de correus `urgent/no urgent`, `important/no important`, etc. a partir de les dades d'un usuari.
  - Fer un classificador de `favorable/contrari/neutre` per les opinions sobre un producte/servei.
  - Verificar l'autor d'un text a partir del seu estil.
  - Fer traducció automàtica de frases d'una llengua `f` a una llengua `e`:

$$best = argmax_e P(e|f) = argmax_e P(f|e)P(e) \tag{7}$$

on $P(e)$ és el model de llenguatge per la llengua e, que s'estima a partir d'un n-grama, i $P(f|e)$ és el model de traducció, que s'estima a partir d'un corpus bilingüe.