TIPE :

AUTOCOMPLETION

MP 2016 - 2017

SOMMAIRE

I - PRESENTATION DU TIPE

DEFINITIONS

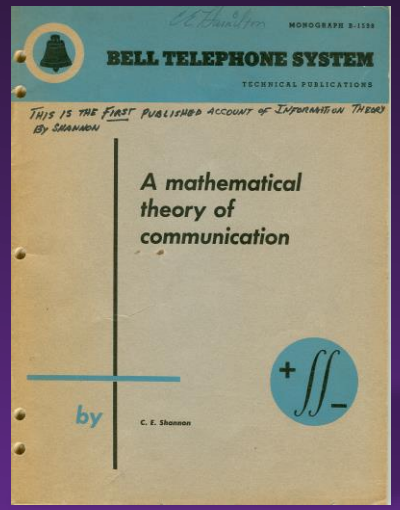
- **Texte** : suite de caractères (ici, éléments de l'alphabet latin ou de sa ponctuation) découpée en propositions (séparées par la ponctuation), elles-mêmes découpées en mots.
- **Autocomplétion** : prédiction d'un mot d'un texte en fonction de ce qui le précède

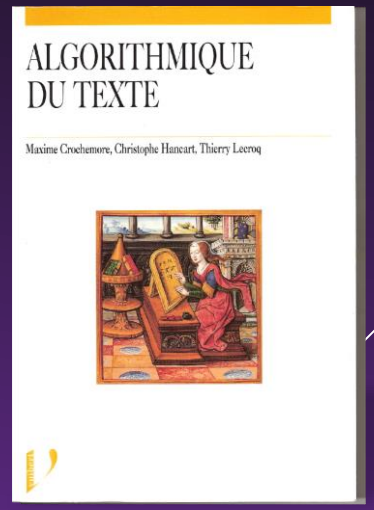
$$\mathcal{A} = \{\mathrm{a}, \mathrm{b}, \dots, \mathrm{z}, \acute{\mathrm{e}}, \grave{\mathrm{e}}, \dots\} \qquad \mathfrak{D} = \{\mathrm{a}, \grave{\mathrm{a}}, \mathrm{abaissa}, \dots, \mathrm{zythums}\} \qquad \mathfrak{P} = \{, . \, ? \, ! : \dots \, " - \}$$

> Sous une forme écrite ou vocale, le texte est le seul véhicule fiable des concepts abstraits.
>
> *Algorithmique du texte*, Crochemore, Hancart, Lecroq

CONTEXTE

- L'essor de technologies appelle celui des théories
- Le développement de la théorie de l'information est donc capital
- Pourtant l'autocomplétion est peu étudiée





OBJECTIFS

- Écrire un programme Python d'autocomplétion en cours de saisie à partir d'une base de données
- Constituer automatiquement cette base de données
- Étudier et optimiser le programme
- Étudier les performances théoriques

II - REALISATION

ORGANISATION

- Dans un premier temps, on compile des informations sur la succession des mots avec le programme « *compiler* »
- Ces informations seront stockées dans un fichier qui fait office de base de données
- Ensuite, on exploite cette base avec le programme principal, « *filler* »

STRUCTURE DE LA LANGUE

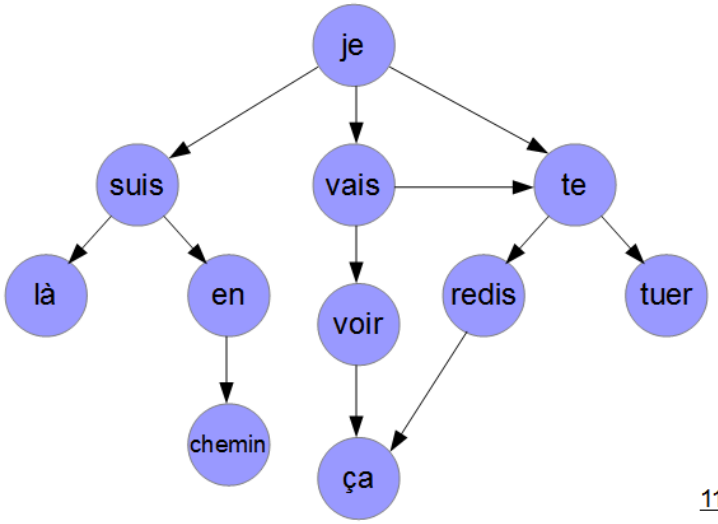


Figure 1 : Modélisation d'une phrase par une chaîne de Markov (source sans mémoire)

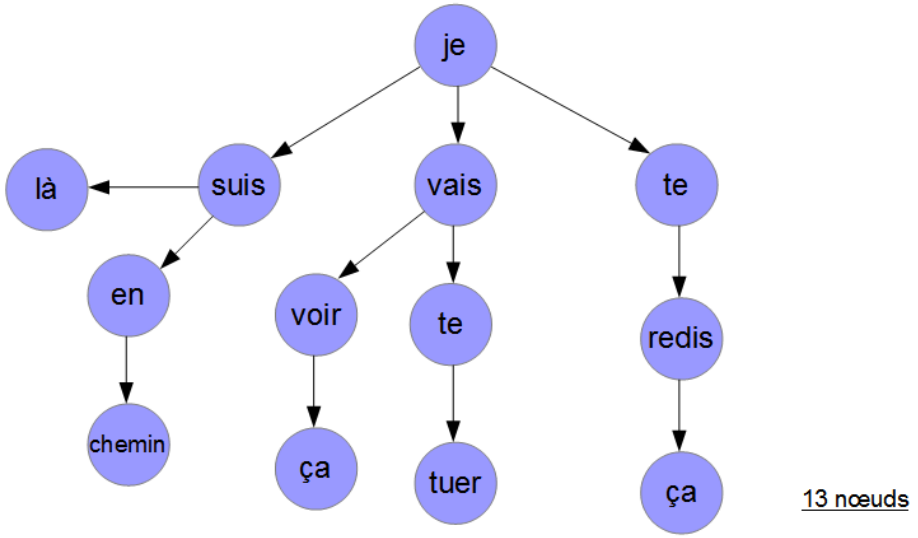


Figure 2 : Modélisation d'une phrase par une chaîne quelconque (source avec mémoire)

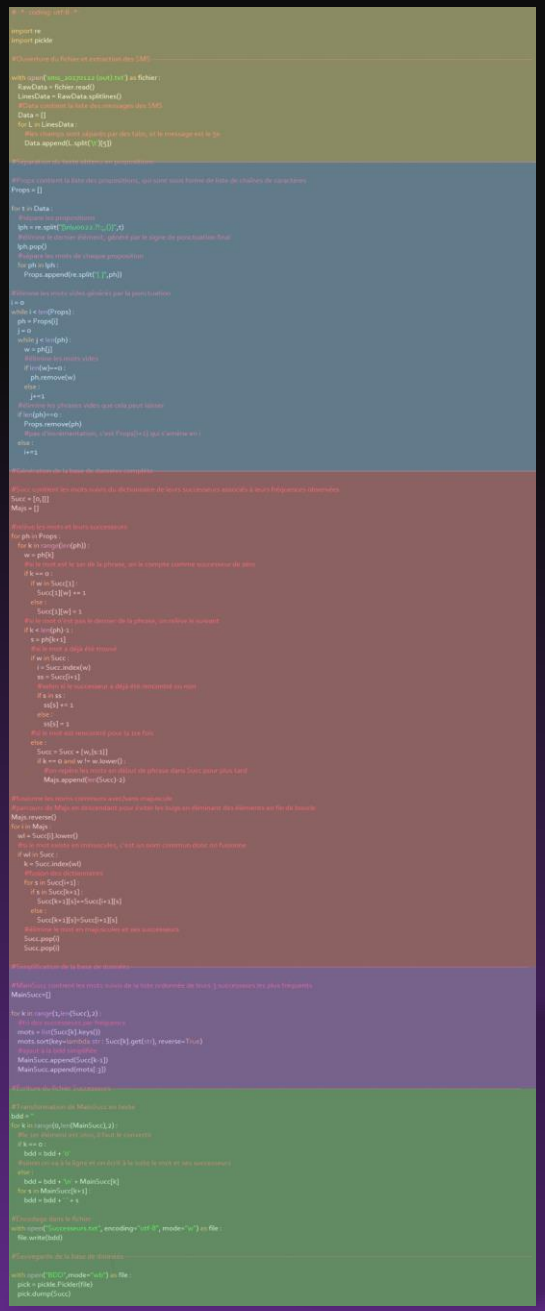
COMPILER

```
# -*- coding: utf-8 -*-

import re
import pickle

#Ouverture du fichier et extraction des SMS

with open('sms_20170222 (out).txt') as fichier :
    RawData = fichier.read()
    LinesData = RawData.splitlines()
    #Data contient la liste des messages des SMS
    Data = []
    for L in LinesData :
        #les champs sont séparés par des tabs, et le message est le 5e
        Data.append(L.split('\t')[5])

#Séparation du texte obtenu en propositions

#Props contient la liste des propositions, qui sont sous forme de liste de chaînes de caractères
Props = []

for t in Data :
    #sépare les propositions
    lph = re.split('[\n\u2026.!?;,()]',t)
    #élimine le dernier élément, généré par le signe de ponctuation final
    lph.pop()
    #sépare les mots de chaque proposition
    for ph in lph :
        Props.append(re.split(' ',ph))

#élimine les mots vides générés par la ponctuation
i = 0
while i < len(Props) :
    ph = Props[i]
    j = 0
    while j < len(ph) :
        w = ph[j]
        #élimine les mots vides
        if len(w)==0 :
            ph.remove(w)
        else :
            j+=1
    #élimine les phrases vides que cela peut laisser
    if len(ph)==0 :
        Props.remove(ph)
        #pas d'incrémentation, c'est Props[i+1] qui s'amène en i
    else :
        i+=1

#Génération de la base de données complète

#Succ contient les mots suivis du dictionnaire de leurs successeurs associés à leurs fréquences observées
Succ = [0,[]]
Majs = []

#relève les mots et leurs successeurs
for ph in Props :
    for k in range(len(ph)) :
        w = ph[k]
        #si le mot est le 1er de la phrase, on le compte comme successeur de zéro
        if k == 0 :
            if w in Succ[1] :
                Succ[1][w] += 1
            else :
                Succ[1][w] = 1
        #si le mot n'est pas le dernier de la phrase, on relève le suivant
        if k < len(ph)-1 :
            s = ph[k+1]
            #si le mot a déjà été trouvé
            if w in Succ :
                i = Succ.index(w)
                ss = Succ[i+1]
                #selon si le successeur a déjà été rencontré ou non
                if s in ss :
                    ss[s] += 1
                else :
                    ss[s] = 1
            #si le mot est rencontré pour la 1re fois
            else :
                Succ = Succ + [w,[s:1]]
                if k == 0 and w != w.lower() :
                    #on repère les mots en début de phrase dans Succ pour plus tard
                    Majs.append(len(Succ)-2)

#Fusionne les noms communs avec/sans majuscule
#parcours de Majs en descendant pour éviter les bugs en éliminant des éléments en fin de boucle
Majs.reverse()
for i in Majs :
    wl = Succ[i].lower()
    #si le mot existe en minuscules, c'est un nom commun donc on fusionne
    if wl in Succ :
        k = Succ.index(wl)
        #fusion des dictionnaires
        for s in Succ[i+1] :
            if s in Succ[k+1] :
                Succ[k+1][s]+=Succ[i+1][s]
            else :
                Succ[k+1][s]=Succ[i+1][s]
        #élimine le mot en majuscules et ses successeurs
        Succ.pop(i)
        Succ.pop(i)

#Simplification de la base de données

#MainSucc contient les mots suivis de la liste ordonnée de leurs 3 successeurs les plus fréquents
MainSucc=[]

for k in range(1,len(Succ),2) :
    #tri des successeurs par fréquence
    mots = list(Succ[k].keys())
    mots.sort(key=lambda str : Succ[k].get(str), reverse=True)
    #ajout à la bdd simplifiée
    MainSucc.append(Succ[k-1])
    MainSucc.append(mots[:3])

#Ecriture du fichier Successeurs

#Transformation de MainSucc en texte
bdd = ''
for k in range(0,len(MainSucc),2) :
    #le 1er élément est zéro, il faut le convertir
    if k == 0 :
        bdd = bdd + '0'
    #ensuite on va à la ligne et on écrit à la suite le mot et ses successeurs
    else :
        bdd = bdd + '\n' + MainSucc[k]
    for s in MainSucc[k+1] :
        bdd = bdd + ' ' + s

#Encodage dans le fichier
with open("Successeurs.txt", encoding="utf-8", mode="w") as file :
    file.write(bdd)

#Sauvegarde de la base de données

with open("BDD",mode="wb") as file :
    pick = pickle.Pickler(file)
    pick.dump(Succ)
```

← Entrée : texte au format .txt

← Sortie : bases de données complète et simplifiée

COMPILER



LECTURE

SÉPARATION

COMPTE

SIMPLIFICATION

ÉCRITURE

FILLER

```
# -*- coding: utf-8 -*-

#Ouverture du fichier et acquisition des données----------
with open('Successeurs.txt', encoding="utf-8", mode="r") as fichier:
    arbre = fichier.read()
    noeuds = arbre.splitlines()
    successeurs = []
    #"successeurs" est une liste de listes, contenant chacune un mot et ses successeurs
    for i in range(0, len(noeuds)) :
        successeurs.append(noeuds[i].split())

#Fonction de succession----------
#renvoie la liste des mots suivant probablement w, par ordre de probabilité
def succ(w) :
    if type(w) != str :
        print('ERREUR : vous devez entrer une chaîne de caractères !')
    else :
        #si début de phrase, file = première liste
        if w in "[\n\u0022.?!:,,«»–(—)–]" :
            file = successeurs[0][-3:]
        else :
            k=1
            #file sera renvoyé tel quel si la boucle se termine sans l'avoir modifiée
            file = ['Aucun successeur trouvé']
            while k in range(1, len(successeurs)) and file == ['Aucun successeur trouvé'] :
                #évite de modifier successeurs
                noeud = successeurs[k].copy()
                #quand on trouve le mot recherché
                if noeud[0] == w :
                    #pour ne garder que les successeurs et pas w
                    noeud.remove(w)
                    file = noeud
                k=k+1
        return(file)

#Saisie intuitive----------

saisie = ['.']
#condition permettant d'arrêter la saisie si besoin
running = True
while running :
    #affichage des suggestions
    S = succ(saisie[-1].lower())
    print(S)
    i = input('Tapez 1, 2 ou 3 (0 si mot absent)\n')
    if i == 'FinDuTest' :
        running = False
    #si le mot voulu est proposé
    elif int(i) != 0 :
        saisie.append(S[int(i)-1])
    #sinon on laisse l'utilisateur le taper
    else :
        saisie.append(input())
```

FILLER

```
# -*- coding: utf-8 -*-

#Ouverture du fichier et acquisition des données
with open('Successeurs.txt', encoding="utf-8", mode="r") as fichier:
    arbre = fichier.read()
    noeuds = arbre.splitlines()
    successeurs = []
    #"successeurs" est une liste de listes, contenant chacune un mot et ses successeurs
    for i in range(0, len(noeuds)) :
        successeurs.append(noeuds[i].split())

#Fonction de succession
#renvoie la liste des mots suivant probablement w, par ordre de probabilité
def succ(w) :
    if type(w) != str :
        print('ERREUR : vous devez entrer une chaîne de caractères !')
    else :
        #si début de phrase, file = première liste
        if w in "[\n\u0022.?!:;,«»–(—)–]" :
            file = successeurs[0][-3:]
        else :
            k=1
            #file sera renvoyé tel quel si la boucle se termine sans l'avoir modifiée
            file = ['Aucun successeur trouvé']
            while k in range(1, len(successeurs)) and file == ['Aucun successeur trouvé'] :
                #évite de modifier successeurs
                noeud = successeurs[k].copy()
                #quand on trouve le mot recherché
                if noeud[0] == w :
                    #pour ne garder que les successeurs et pas w
                    noeud.remove(w)
                    file = noeud
                k=k+1
        return(file)

#Saisie intuitive

saisie = ['.']
#condition permettant d'arrêter la saisie si besoin
running = True
while running :
    #affichage des suggestions
    S = succ(saisie[-1].lower())
    print(S)
    i = input('Tapez 1, 2 ou 3 (0 si mot absent)\n')
    if i == 'FinDuTest' :
        running = False
    #si le mot voulu est proposé
    elif int(i) != 0 :
        saisie.append(S[int(i)-1])
    #sinon on laisse l'utilisateur le taper
    else :
        saisie.append(input())
```

LECTURE

FONCTION DE SUCCESSION

INTERFACE

RESULTATS

- En testant *filler*, le programme propose le mot attendu dans 17,3% des cas

Je ne pense pas changer l'histoire de ce domaine car il y a eu beaucoup de grands hommes ici avant moi, et je ne travaille pas dessus depuis assez longtemps.

III - AMELIORATION

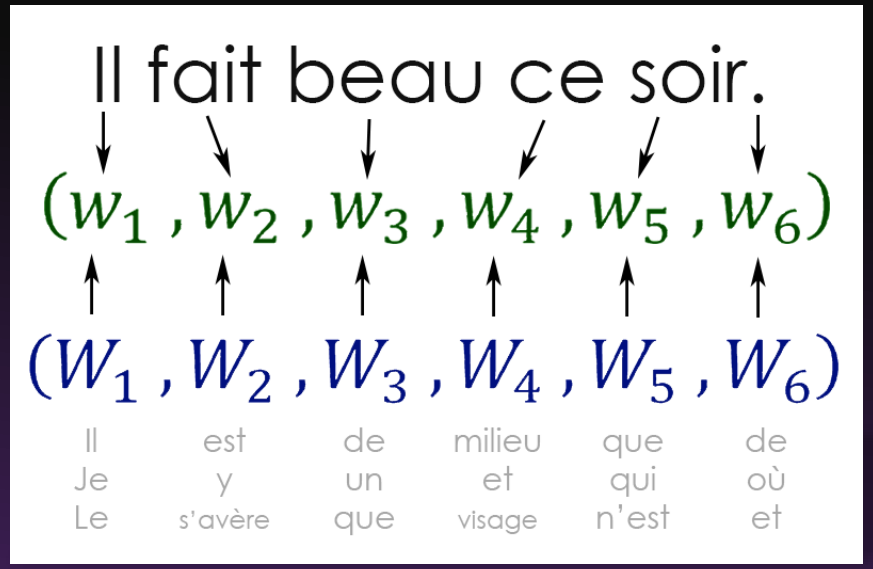
STRUCTURE DE LA BASE DE DONNEES

- Afin d'accélérer le parcours de la base de données, on la trie dans l'ordre alphabétique
- On doit donc également modifier *filler* en implémentant une recherche par dichotomie
- Un tri par fréquence d'apparition serait possible, mais moins intéressant au final

IV - ETUDE THEORIQUE

COMPLEXITE

- Avec n le nombre de mots enregistrés, la complexité de *filler* est en $O(n)$ pour le fichier non trié, et en $O(\log(n))$ une fois l'amélioration effectuée.



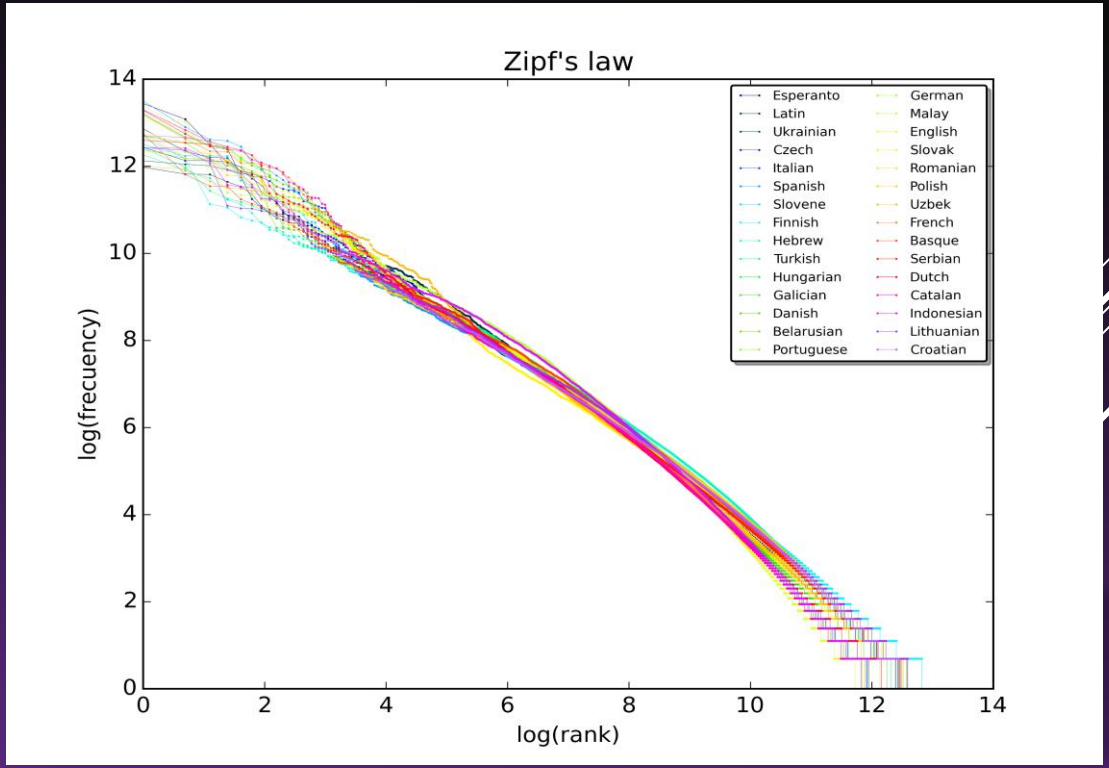
MODELISATION DU LANGAGE

- Soit $(w_k)_{k\in[\![1;n]\!]}$ une phrase et $(W_k)_{k\in[\![1;n]\!]}$ la famille de variables aléatoires correspondante.
- On cherche la probabilité $\mathrm{P}(W_k = w_k | W_{k-1} = w_{k-1})$

LOI DE ZIPF

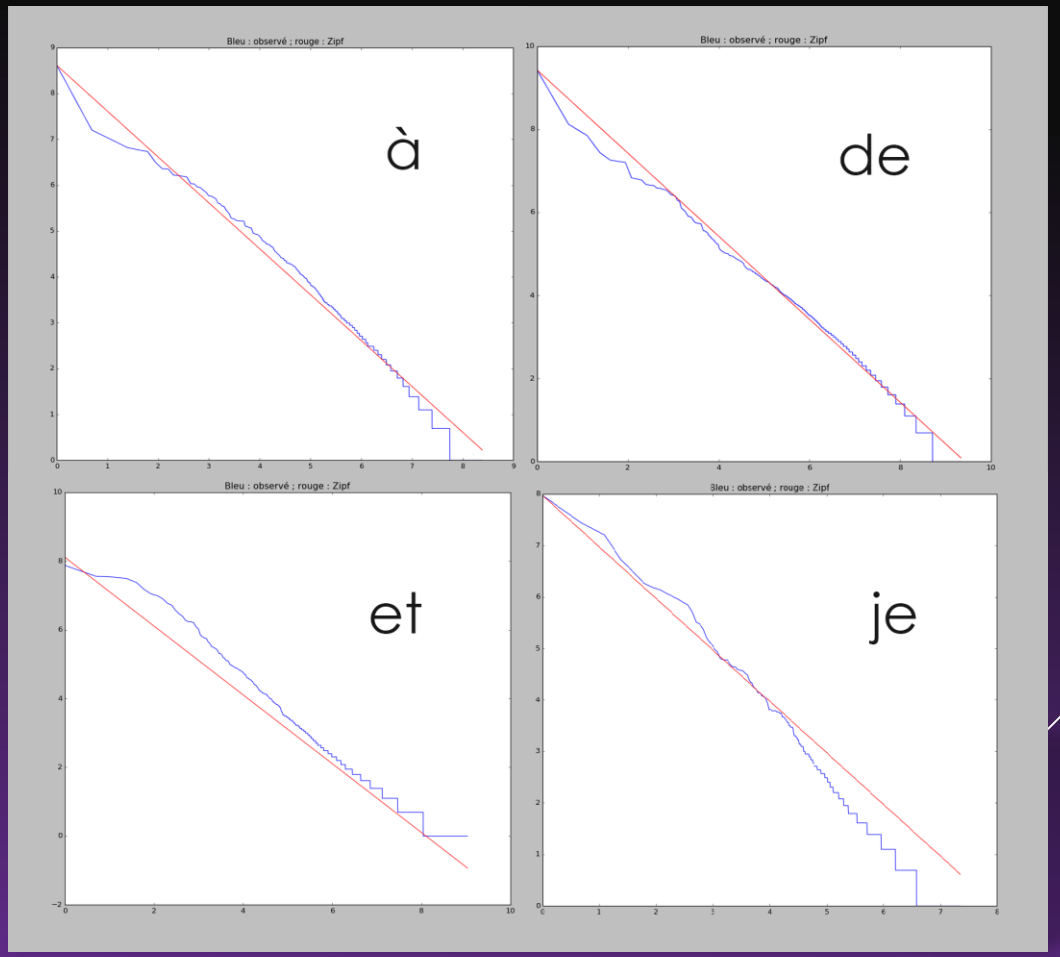
- La répartition des mots suit la loi de Zipf : en notant m_j le j[e] mot le plus courant, on a :

$$P(W_k = m_j) = \frac{P(W=m_1)}{j} = \frac{1}{j\mathcal{S}_N} \qquad \text{où } \mathcal{S}_N = \sum_{k=1}^{N} \frac{1}{k} \quad \text{avec } N = \text{card}(W_k(\Omega))$$



LOI DE ZIPF

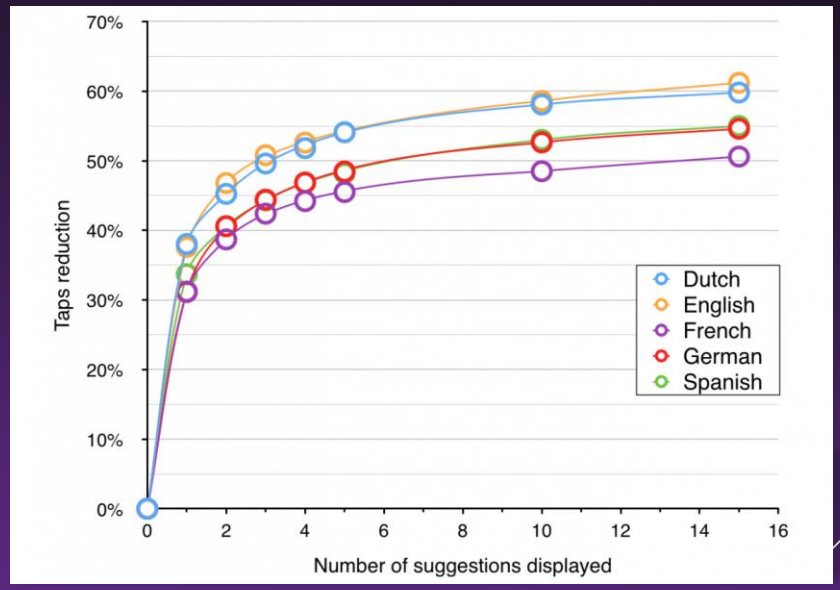
- Expérimentalement, les successeurs d'un mot donné suivent aussi une loi de Zipf



PROBABILITES

- D'où $\mathrm{P}(W_k = m_j | W_{k-1} = w_{k-1}) = \frac{1}{j\mathcal{S}_N}$, où N est le nombre de successeurs possibles
- Notant $\mathrm{S} = \{s_1; s_2; s_3\}$ le triplet de suggestions affichées, on trouve en moyenne :

$$P(W_k \in \mathrm{S}) = \left(1 + \frac{1}{2} + \frac{1}{3}\right)\frac{1}{\mathcal{S}_N} \approx 49{,}7\%$$



V - CONCLUSION

BILAN

- Nous avons bien créé un programme d'autocomplétion fonctionnel
- Toutefois ses capacités ne sont pas encore optimales

PISTES D'AMELIORATION

- Élargir la base de données peut permettre de meilleures prédictions
- Utiliser une variante du modèle de Zipf
- Prendre une modèle de Markov d'ordre 2