Szövegmetrikák

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.2	2011.11.13	MatMod szeminárium	KF

Tartalomjegyzék

1 Natural Language Toolkit 2

2 Távolság mértékek 3

3 Hasonlóság mértékek 5

4 LCS hossz alapú hasonlóság 6

5 Bináris távolság 6

6 Hamming távolság 7

7 Levenshtein távolság 7

8 Levenshtein modul 8

9 Jaro hasonlóság 8

10 Jaro-Winkler hasonlóság 9

11 Soundex kontrakció 10

12 Egy kis szakirodalom 11

13 Köszönöm a figyelmet 12

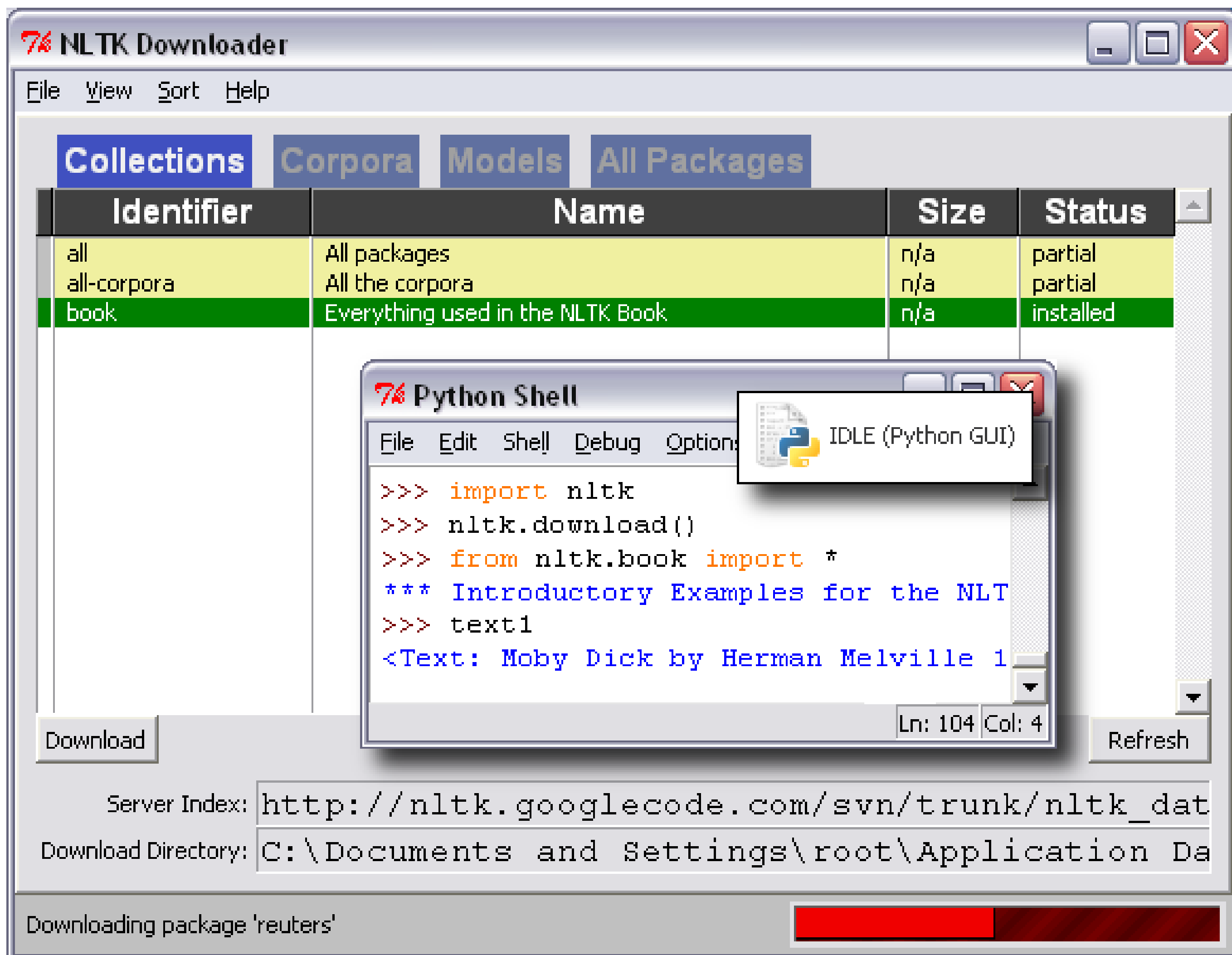
14 NLTK érdekességek 12

“Elárasztanak minket az adatok — tudományos, egészségügyi, demográfiai, pénzügyi és marketing adatok. Az embereknek nincs idejük mindezt értelmezni, de már elolvasni sem. Az emberi figyelem az egyik legértékesebb erőforrássá vált.”

— Microsdoft research *Jim Gray*

1. Natural Language Toolkit

Az NLTK a Python legkomplexebb, természetes nyelvű szövegeket feldolgozó csomagja. Funkcionalitása számos osztályra van bontva:



1. ábra. Indítsuk el az IDLE-t, és töltsünk le néhány szöveggyűjteményt. . .

- `nltk.corpus` : szabványos természetes nyelvű szövegek korpuszkezelője
- `nltk.tokenize`, `nltk.stem` : szavakra bontó és szótövező elemek
- `nltk.tag` : part-of-speech (POS) jelölés
- `nltk.classify`, `nltk.cluster`: felügyelt és nem felügyelt osztályozó eszközök
- `nltk.metrics` : többek között szövegmetrika
- és számos egyéb: `nltk.collocations`, `nltk.chunk`, `nltk.parse`, `nltk.sem`, `nltk.inference`, `nltk.probabili` `nltk.app`, `nltk.chat`

2. Távolság mértékek

Amint a szövegekből matematikai struktúrákat tudunk készíteni, minden hagyományos modell és eszköz használható. Pl. a szöveg-dokumentum *mátrix*, szöveg távolság *mértékek*, stb.

Metrics

Azonos típusú elemek között értelmezett *távolságfüggvény*, amelyik eleget tesz az alábbi feltételeknek:

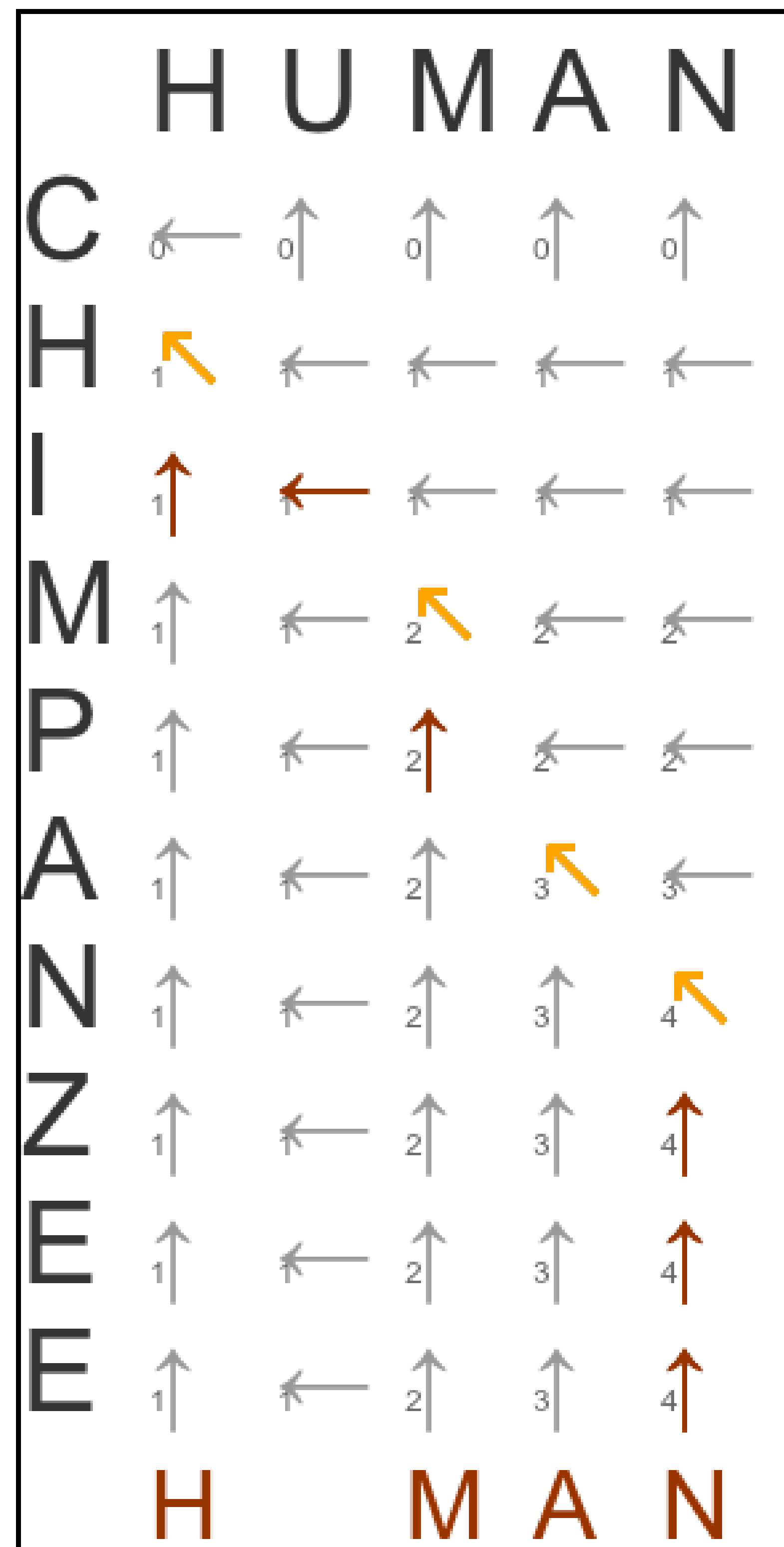
- d(a,a) = 0 : azonosság
- d(a,b) $\geq$ 0 : nemnegativitás
- d(a,b) + d(b,c) $\geq$ d(a,c) : háromszög-egyenlőtlenség

- d(a,b) = d(b,a) : szimmetria
  (szövegek esetében nem kötelező!)

Contraction

Közelítő transzformáció, melynek során eredetileg különböző elemeket azonosnak tekintünk. Szövegek esetében ilyen például a kis- és nagybetűk megkülönböztetésének mellőzése, vagy ékezetes betűk esetén az ékezettelenítés.

3. Hasonlóság mértékek



Similarity

Azonos típusú elemek között értelmezett *hasonlóságfüggvény*. Általában metrikából származtatható. Főbb tulajdonságai:

- s(a,a) = 1 : azonosság

- $0 \leq$ s(a,b) ≤ 1 : standardizált
- s(a,b) $\leq$ s(a,c) $\therefore$ d(a,b) $\geq$ d(a,c) : monotonitás
  (szövegek esetében nem kötelező!)
- s(a,b) = s(b,a) : szimmetria
  (szövegek esetében nem kötelező!)

Longest Common Subsequence (LCS)
A leghosszabb közös szekvencia az a string, amelyik mindegyik szövegben megtalálható. Hasonlóság számításra is felhasználható, de elsősorban bioinformatika alkalmazásai vannak. Többértékű függvény, hiszen két szövegnek több különböző leghosszabb közös része is lehet.

- *LCS('human', 'chimpanzee') = 'hman'*

4. LCS hossz alapú hasonlóság

```
def lcs_len(S, T):
    m = len(S); n = len(T)
    L = [[0] * (n+1) for i in xrange(m+1)] # list enchanment - a csoda!
    lcs = 0
    for i in xrange(m):                    # range() enchanment
        for j in xrange(n):
            if S[i] == T[j]:
                L[i+1][j+1] = L[i][j] + 1
                lcs = max(lcs, L[i+1][j+1])
    return lcs

>>> 2.0 * lcs_len('Gyurcsány','Orbán') / len('Gyurcsány'+'Orbán')
# 0.375   # Fontos: ha 2.0 helyett csak 2-est írunk, 0-át kapunk!
```

Szöveg műveletek és függvények

- + : a szövegösszefűzés (konkatenáció) jele
- len() : szöveg hosszának |*s*| kiszámítása (utf8 kompatibilitás?)

5. Bináris távolság

Bináris távolság
Vagy megegyezik a két szöveg, vagy nem.

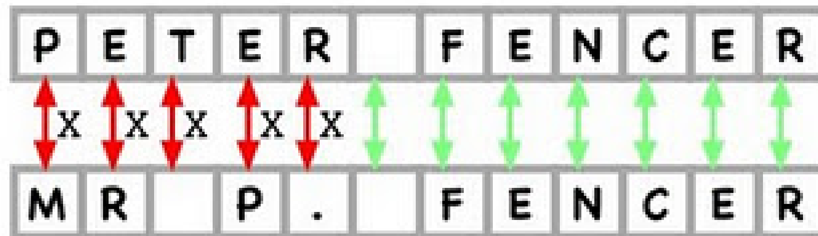
```
import nltk
nltk.metrics.demo()
help(nltk.metrics.distance)

help(nltk.metrics.distance.binary_distance)
# Help on function binary_distance in module nltk.metrics.distance:
#
# binary_distance(label1, label2)
#     Simple equality test.
#
#     0.0 if the labels are identical, 1.0 if they are different.
#
#     >>> binary_distance(1,1)
#     0.0
#
```

```
#     >>> binary_distance(1,3)
#     1.0

nltk.metrics.distance.binary_distance('Gyurcsány', 'Orbán')
# 1.0
```

6. Hamming távolság



Hamming távolság

A legegyszerűbb távolság, a pozícionálisan egyező karakterek számát adja meg. Eredetileg csak azonos hosszú szövegek esetére definiálták.

```
def hamming_distance(s1, s2):
    assert len(s1) == len(s2)
    return sum(ch1 != ch2 for ch1, ch2 in zip(s1, s2))

>>> hamming_distance('Gyurcsány', 'Orbán')
# Traceback (most recent call last):
#   File "<pyshell#13>", line 1, in <module>
#     hamming_distance('Gyurcsány', 'Orbán')
#   File "<pyshell#12>", line 2, in hamming_distance
#     assert len(s1) == len(s2)
# AssertionError
```

A `zip()` függvény csak addig adja vissza a két lista elemeit, ameddig az egyik el nem fogy.

---

megjegyzés

Különböző hosszúságú szövegek esetén vagy \0 karakterrel kell feltöltést végezni, vagy a rövidebbik szerint csonkolni kell, és hozzáadni a különbség hosszát.

---

7. Levenshtein távolság

Levenshtein távolság

Szerkesztési távolság, ahol egy karakter törlése, beszúrása vagy lecserélése egy-egy "hibapontnak" számít. A megoldó algoritmus egy *dinamikus programozási* módszert alkalmaz, amit önrekurzióval valósítottunk meg.

```
def levenshtein_distance(a, b):
    if not a: return len(b)
    if not b: return len(a)
    return min( \
      levenshtein_distance(a[1:], b[1:])+(a[0] != b[0]), \
```

```
        levenshtein_distance(a[1:], b)+1, \
        levenshtein_distance(a, b[1:])+1 \
        )

>>> levenshtein_distance('Gyurcsány', 'Orbán')
```

Slicing

A listák és tömbök végtelenül egyszerű darabolási technikája *[:]*.
A szövegek tehát karaktersorozatból álló objektumok, ahol az első karakter indexe *1*.

Az NLTK-ban is megtalálható a metrikák között

```
nltk.metrics.edit_distance ('Gyurcsány','Orbán')
# ?
```

8. Levenshtein modul

Azért vannak olyan algoritmusok, amelyeket más nyelven írtak és fordítottak a python alá, hogy kellően gyorsak legyenek. Így van ez a Levenshtein távolsággal is.

```
import Levenshtein
help(Levenshtein)

Levenshtein.hamming('Gyurcsány','Orbán')
# Traceback (most recent call last): ...
# ValueError: hamming expected two strings of the same length

Levenshtein.distance('Gyurcsány','Orbán')
# 6
```

De van benne számos egyéb speciális távolság és hasonlóság függvény is.

```
# Minimális szerkesztési távolságon alapuló hasonlóság
Levenshtein.ratio('Gyurcsány', 'Orbán')
# 0.5

# Több szöveg között a centroid
Levenshtein.median(['SpSm', 'mpamm', 'Spam', 'Spa', 'Sua', 'hSam'])
# 'Spam'

# Magyar ékezetekkel az UTF-8 miatt még van egy kis tennivaló
Levenshtein.median(['Gyurcsány', 'Orbán'])
# 'GOrbcs\xc3\xa1n'            # GOrbcsán
Levenshtein.quickmedian(['Gyurcsány', 'Orbán'])
#'Gyrcs\xa1nn'                 # Gyrcsánn
```

9. Jaro hasonlóság

Jaro hasonlóság

Az *egyező* betűk számát, illetve a *betűfelcseréléseket* veszi figyelembe a távolság számításakor. A lényege, hogy akkor hasonlít jobban két szöveg egymáshoz, ha a *rész betűsorozatai* megfeleltethetőek egymásnak.

Jelölések: $|s1|$, $|s2|$ a szövegek hossza, m a megegyező betűk száma, t a felcserélt betűk száma

$$d_j(s_1, s_2) = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$$

Ha az így számított távolság nagyobb, mint a hosszabbik szöveg fele, akkor a két szöveget teljesen eltérőnek vesszük.

---

megjegyzés

- Előnye, hogy *rövid* szövegekre lett kifejlesztve, azon belül is *személynevekre*. Ráadásul futási időben hasonlítható a dinamikus megoldásokhoz.
- Hátránya, hogy matematikai értelemben véve *nem ad metrikát*, mert nem teljesül rá a háromszög egyenlőtlenség. Tehát azok az osztályozók, amelyek a távolság eme tulajdonságát használják ki, félreosztályozhatnak.

```
Levenshtein.jaro('Brian', 'Jesus')
# 0.0
Levenshtein.jaro('Thorkel', 'Thorgier')
# 0.779761904761904 77
Levenshtein.jaro('Dinsdale', 'D')
# 0.7083333333333337

Levenshtein.jaro('Gyurcsány','Orbán')
# 0.6888888888888889
```

10. Jaro-Winkler hasonlóság

Jaro-Winkler hasonlóság

A Jaro algoritmus Winkler-féle továbbfejlesztése, amelyik azon alapul, hogy a szavak *elején* és *végén* a betűsorrend helyessége sokkal *fontosabb* az azonosításkor, mint a közepén. Így a közepén történő különbségeket kisebb súllyal veszi figyelembe. A Winkler-féle szélsúlyozás esetén meg kell állapítani a két szöveg elején (illetve végén) a *közös prefixet* (postfixet).

Jelölések: l a prefix (postfix) hossza, $p < 0.25$ skálázási faktor, (ami alapértelmezés szerint $p = 0.1$).

$$d_w = d_j + l \cdot p(1 - d_j)$$

---

megjegyzés

Arocdnicg to rsceearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer are in the rghit pcale. The rset can be a toatl mses and you can sitll raed it wouthit pobelrm. Tihs is buseace the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

```
Levenshtein.jaro_winkler('Brian', 'Jesus')
# 0.0                    # jaro: 0.0
Levenshtein.jaro_winkler('Thorkel', 'Thorgier')
# 0.867857142857142 88    # jaro: 0.779761904761904 77
Levenshtein.jaro_winkler('Dinsdale', 'D')
#  0.7375000000000004    # jaro: 0.7083333333333337

Levenshtein.jaro_winkler('Thorkel', 'Thorgier', 0.25)
# 1.0

Levenshtein.jaro_winkler('Gyurcsány','Orbán')
# 0.6888888888888889      # jaro: 0.6888888888888889
```

11. Soundex kontrakció

Soundex
: Angol szavakat közelít egymáshoz az angol nyelv szavainak kiejtés szerinti hasonlósága alapján. Minden szóhoz egy négy hosszú (1 betű + 3 szám) szöveget rendel.

Nyílt forráskódú rendszerekben azért nincs benne alapértelmezettként, mert a szerzők (Robert C. Russell és Margaret K. Odell) 1918-ban és 1922-ben is levédették.

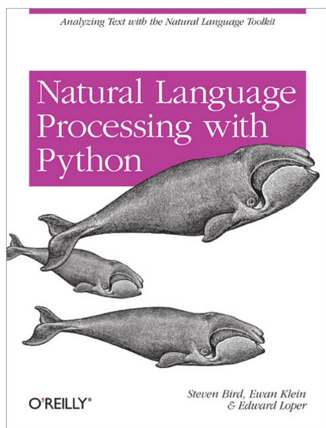
```
import re

def soundex_prepare(s):
    """Prepare string for Soundex encoding.
    Remove non-alpha characters (and the not-of-interest W/H/Y),
    convert to upper case, and remove all runs of repeated letters."""
    p = re.compile("[^a-gi-vxz]", re.IGNORECASE)
    s = re.sub(p, "", s).upper()
    for c in set(s):
        s = re.sub(c + "{2,}", c, s)
    return s

def soundex_encode(s):
    """Encode a name string using the Soundex algorithm."""
    result = s[0].upper()
    s = soundex_prepare(s[1:])
    letters = 'ABCDEFGIJKLMNOPQRSTUVXZ'
    codes   = '.123.12.22455.12623.122'
    d = dict(zip(letters, codes))
    prev_code=""
    for c in s:
        code = d[c]
        if code != "." and code != prev_code:
            result += code
        if len(result) >= 4:
            break
        prev_code = code
    return (result + "0000")[:4]

soundex_encode(soundex_prepare('Smith'))
# 'S530'
soundex_encode(soundex_prepare('Smythe'))
# 'S530'
```

12. Egy kis szakirodalom



Quick answers to common problems

Python Text Processing with NLTK 2.0 Cookbook

Over 80 practical recipes for using Python's NLTK suite of libraries to maximize your Natural Language Processing capabilities

Jacob Perkins

[PACKT] open source
PUBLISHING

13. Köszönöm a figyelmet

tipp
Melyik a második leggyakoribb szó az angolban, amelyik **f**-fel kezdődik és **k**-val végződik?

```
>>> import nltk
>>> from nltk.book import *

>>> [w for w in text4 if w.startswith('f') and w.endswith('k')]
['frank', 'frank', 'framework', 'frank', 'framework', 'framework']
>>> set([w for w in text4 if w.startswith('f') and w.endswith('k')])
set(['frank', 'framework'])

>>> fdist1 = FreqDist([w for w in text3 if re.match('^f.*k$',w)])
>>> sorted(fdist1)
[u'flock', u'folk']

>>> sorted(FreqDist([w for w in text6 if re.match('^f.*k$',w)]))
['folk', 'footwork']
```

14. NLTK érdekességek

Gyakoroljunk!
Bűn és az ő bűhődése pythonban. . .

```
import nltk
from nltk import *
from urllib import urlopen

# import something from a webpage
```

```
url = "http://www.gutenberg.org/files/2554/2554.txt"
html = urlopen(url).read()
# convert html to plain text format
raw = nltk.clean_html(html)
# tokenize and get ready for use
tokens = nltk.word_tokenize(raw)
text = nltk.Text(tokens)

# concordance
text.concordance("Sonia")
# similarity
text.similar("monstrous")
# common contexts
text.common_contexts(["Sonia", "Sofya"])
# times mentioned
text.dispersion_plot(["Raskolnikov", "Semyonovna", "Razumihin"])
```