EDAN20 Language Technology
2017-08-29

Rasmus Olofzon
muh11rol

Lab 0

Introduction

This report is the hand-in of a compulsory part of the zero:th lab in the course Language Technology (EDAN20) 2017/2018 at the Faculty of Engineering at Lund University. It is essentially comments on a spell checker written in Python by Peter Norvig [1].

Comments

```
import re
from collections import Counter
```

Import regex and Counter modules. Counter is a subclass of python's class dict (dictionaries, key-value data structures, e. g. { 'key1':value1, 'key2':value2, … , 'keyN':valueN }) and is used for counting things. It will list objects and their count in a dict like { 'A':5, 'B':3, 'C':2, 'F':1 }, as an example of counting grades in courses.

```
def words(text): return re.findall(r'\w+', text.lower())
```

A function that returns all words, lowercased, from a string *text*. The method re.**findall**(*pattern*, *string*, *flags=0*) finds all unique occurrences that matches *pattern* in *string*. The pattern is specified as a raw string. Prefix r signifies that the following string is a raw string, this means that e. g. r'\x40' == '\\x40' evaluates to True. \w is a special sequence in regular expressions, and means to select "Any word character: letter, digit, or underscore. Equivalent to [a-zA-Z0-9_]" [2], i. e. all 'words' from the *string*. The *string* is in this case the passed parameter, lowercased through python string's lower() method.

```
WORDS = Counter(words(open('big.txt').read()))
```

What happens here is that the file big.txt is opened, its contents read and passed as parameter to the function words(text), which as described above returns all words lowercased from a string, in this case the file's contents. These are then counted with a Counter, which returns and assigns to WORDS a dictionary with all unique words and how many times they occur in the text file's text (hereby referred to as the Dictionary).

```
def P(word, N=sum(WORDS.values())):
    "Probability of `word`."
    return WORDS[word] / N
```

Calculates classical probability of a certain *word* by dividing the value for the *word* in WORDS, which is the number of occurrences of *word* in the Dictionary, with the sum of WORDS' values, which is the number of actual words in the Dictionary.

```
def correction(word):
    "Most probable spelling correction for word."
    return max(candidates(word), key=P)
```

This does what the code comment says, returns the "Most probable spelling correction for word". This happens through passing all candidates for correct spelling of the passed *word* to the max method with the strategy P. What this means is that for all candidates the classical probability/relative frequency of that string in the Dictionary is calculated and the candidate with the highest calculated probability is selected as return value for the function correction(word).

The candidates function is described below. Please note that the functions are from here on not described in the same order as they are written in [1]. This is because they use each other in a way that it makes more sense to describe them in the following order: edits1, edits2, known and candidates.

```
def edits1(word):
    "All edits that are one edit away from `word`."
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in 
letters]
    inserts    = [L + c + R               for L, R in splits for c in 
letters]
    return set(deletes + transposes + replaces + inserts)
```

The function edits1 is based on the edit distance that is described in [1]: that when the edit of one character (adding of, removal of, change of or changing place of) of a string will yield the original string, there is an edit distance of 1 between the original string and the other string. This can be extrapolated to longer edit distances. The function edits2 deals with edit distances of 2.

Here the comprehension principle from page 40 of [3] is used. A *splits* list of tuples is constructed by moving the split point/pivot point through all indices of the *word*. This is accomplished with a for in-loop and the slice functionality of python strings. Slicing works so that str[start:end] will take a substring of *str* starting from the index *start* to *end* – 1. E. g. ‘language’[1:6] == ‘angua’.

The list of strings *deletes* is constructed by looping through the tuples in the *splits* list where the right (R) string of the split is not the empty string and removing the first character of R (actually taking substring of second character to the end). *transposes* is constructed by taking the tuples from *list* and changing place of first and second character of R (where R is longer than one character, if not there is no possibility of changing places). *replaces* is constructed by replacing the first character of R with one character of the alphabet (from *letters*). *inserts* is constructed by adding one alphabet character into the split, in all possible permutations. All the values from these constructions are added to a *set*, which is a python data structure that only adds unique occurrences.

```
def edits2(word):
    "All edits that are two edits away from `word`."
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))
```

Deals with strings with an edit distance of 2 from the original string. Basically all possible (in the transformations allowed by this program) strings with edit distance 1 from *word* are constructed, then all these transformed strings are run through *edits1* again. This yields all transformations with edit distance 1 from the transformed strings, effectively adding another unit of edit distance from *word*. 1 + 1 = 2, thus all strings with edit distance 2 from *word* are constructed and placed in a tuple, and that tuple is returned.

```
def known(words):
    "The subset of `words` that appear in the dictionary of WORDS."
    return set(w for w in words if w in WORDS)
```

Constructs a *set* with all values from *words* that also appear in the keys of *WORDS*. Usage of this function is explained below.

Rasmus Olofzon
muh11rol

```
def candidates(word):
    "Generate possible spelling corrections for word."
    return (known([word]) or known(edits1(word)) or known(edits2(word)) or
[word])
```

Returns the candidates for correct spelling of the passed *word*. It uses a probability hierarchy (also described in [1]), where the most probable candidate is the passed *word* if it also appears verbatim in WORDS (checked through the *known* function). Next step in the hierarchy is if there is a string with edit distance 1 from *word* that also exists in WORDS. Next step is string with edit distance 2 from *word* that also exists in WORDS. If there is no such thing, there is no ‘guess’ for which word that was intended to be written and the actual *word* is returned.

[1] http://norvig.com/spell-correct.html , retrieved 2017-08-29

[2] http://fileadmin.cs.lth.se/cs/Education/EDAN20/Slides/EDAN20_ch02.pdf , retrieved 2017-08-29

[3] Draft of version 3 of the Language processing book by Pierre Nugues