L2X: Spelling correction

Robin Kurtz, Marco Kuhlmann

Goal	Implement a simple spelling corrector based on Levenshtein distance. Understand a dynamic programming algorithm for computing the Levenshtein distance between two strings.
Preparations	Read Section 2.5 (on minimum edit distance and spelling correction) in Jurafsky and Martin (2018) and watch the video material linked on the course website.
Report	Submit the requested code.

01 A **spelling corrector** is a system that detects and corrects spelling errors that result in non-words. To detect non-words, the system tries to look up each token in a dictionary of known words; if this look-up fails, the token is considered a non-word and marked as an error. To correct an error, the system considers a list of candidate corrections: known words that are similar to the non-word. From these the system then picks the most likely candidate, and suggests it as the correction of the non-word.

02 Your task in this assignment is to implement a simple correction component that provides the functionality just described. You will select candidate corrections from the vocabulary of three collections of Sherlock Holmes stories (the same training data as in lab L2), and measure the similarity between a non-word and a candidate using Levenshtein distance.[1] Candidate words with some Levenshtein distance d are considered to be more likely than words with any greater Levenshtein distance $d' > d$. However, several candidates might have the same distance to the non-word, and in these cases you should use word frequency to break the tie, giving preference to the candidate word with the higher frequency. If two words have the same Levenshtein distance *and* frequency, you should pick the word that comes first according to alphabetical order.

[1] In a real spelling corrector, the Damerau–Levenshtein distance would be more appropriate.

03 At the end of this lab you should submit a standalone script `spelling.py` that

1. reads in Sherlock Holmes novels from one or more files;
2. tokenizes the novels, builds up a vocabulary, and counts word frequencies;
3. reads a list of non-words from the standard input;
4. for each non-word, prints out the most likely candidate word in the novels.

04 To implement the script, you are given a minimal skeleton file in

```
/courses/729G17/labs/l2x/code/
```

The file contains a function `tokens()` that takes care of tokenization and returns an iterator over the tokens in a file-like data source. The remaining code should be written by yourself. When implementing the Wagner–Fischer algorithm for computing the Levenshtein distance between a non-word and the candidate correction, you may take inspiration from other implementations (available in the course book or on the internet), but you should still be able to explain every single line of your code. You are not allowed to use a library.

05 For testing purposes, you are given the file

```
/courses/729G17/labs/l2x/data/test.txt
```

This file contains non-words and their intended corrections in a simple two-column format, with columns separated by tab characters, like so:

```
intention    execution
```

With this test file, it should be possible to call your script as follows:

```
python3 spelling.py advs.txt mems.txt retn.txt < test.txt
```

Thus the names of the files containing the novels should be given to the script as command line arguments and the candidates should be read from standard input. The output produced by your script should follow the same format as the test file such that your script will print out the same pairs that are given in the test file.

06 When building up the vocabulary and counting the word frequencies you can simplify your code by using the `Counter` class from Python's `collections` module.