Nile University
School of Engineering and Applied Sciences
Electronics and Computer Engineering (ECE)

NLP (ECEN 550)

Spring 2020

Course Project

Text editor with error correction & regex search

Submitted by: **Gad Mohamed Gad**, ID: **1710341**, Email: **g.gad@nu.edu.eg**

Submitted to: Dr. Nermin Negied

31/5/2020

Table of Contents

Abstract

Spell checking is one the most widely used tasks in NLP in broad range of applications like information retrieval, proofreading, and text editors. Spell checking is a multi-phase task where it first break down the text for spelling errors establish an algorithm to find the correct word the user meant and replace it directly or inform the user to choose the appropriate word if the algorithm is not confident about its choice. In this project, a text editor is built with minimal features to show spell checking and error correction in action. Also, Regex search is added to enable the user to search within the text using regex expressions.

Introduction

Overview on spell checking

The Frequency of spelling errors in human typed text varies from:

1- 0.05% of the words in carefully edited newswire, to
2- 38% in difficult applications like telephone directory lookup.

There are two types of spelling errors:

1- Non-word errors
   Where there is a misspelled word (like: moenster -> monster)
2- Real-word errors
   a- Typographical errors (like: three -> there)
   b- Cognitive errors (homophones)
      Words that have the same sound (homophones) but have different meaning (like: piece -> peace, and too -> two)

Real-word errors are much harder to detect and correct, it needs the spell checking engine to be aware of the context of the sentence to detect the Real-word fault and be aware of a bigger context to able to correct this word with a totally different word (have high Levenstien distance). Thus, this application will only consider non-word type of spell error detection and correction.

Non-word Spell error detection

In order to recognize non-word spelling errors, we consider any word that is not in our dictionary to be an error (we can also make the spell checker more efficient by allowing the user to add more words in the dictionary online (while writing) if he is sure that this word is correct, this feature exist in spell checking engines in software like Word). Also, we should have a large dictionary to able to judge the correctness of the word and better to be a specialized dictionary so for example if the spell checking engine is working in an application related to the medical industry, it has to be rich with medical terminology.

Non-word Spell error correction

Non-word spell error correction is a 4 phase process:

1- The first step towards correcting a detected error is to generate ALL possible candidate words (even if most of these possible candidates are not even valid words, they have to be generated and then checked against the dictionary and removed if they are not valid).
2- After generating all possible candidates for the non-word error, we check them against the dictionary to delete candidates that are not valid words.
3- The remaining candidates are valid words that need to be compared with the wrong word in terms of levenshtien distance i.e. how many insertions, deletions, or substitutions are needed for the wrong word detected to be converted to the each and every word in the list of valid candidates.
4- The decision after that on whether to directly substitute the wrong word with the closest candidate (lowest levenshtien distance score) or to propose this closest candidate to the user for him to choose to make the susbstitution or not, or to propose a list of 5 or 10 closest candidates.

HCI confidence levels

These are the conditions on which we decide which decision to take after analyzing the wrong word.

- Autocorrect if very confident (levenshtien distance = 1)
- Propose 1 or more if less confident (many candidates with levenshtien distance = 1 or more)
- Flag as error if not confident (no valid candidates)

The mechanism of choosing the correct replacement for a wrong word can be mathematically described as:

$$\text{Argmax}_{c \in candidates} P(c|w) \qquad (1)$$

Where:

- w is the wrong word
- c is a candidate correction.

According to Bayes' theorem, equation (1) is equivalent to:

$$\text{Argmax}_{c \in candidates} P(c)\, P(w|c) / P(w) \qquad (2)$$

Where:

- P(c) is the language model, i.e. the frequency of the correction candidate appearing in a representative corpus.
- P(*w*|*c*) is the error model, i.e. the probability that when the user meant to type c, he misspelled it with w. For example, P(*teh*|*the*) is relatively high, but P(*theeexyz*|*the*) would be very low.

Since P(w) is the same for all candidate corrections c, we eliminate it:

$$\text{Argmax}_{c \in candidates} P(c)\, P(w|c) \qquad (3)$$

Regex

Regular expressions are an extremely useful technique used for information extraction from any text through searching for one or more expression matches in a search pattern. Applications that use regular expressions range from validation to parsing/replacing strings, and web scraping, or, as in our application, information extraction.

Following are some examples of some regular expressions and how they operate on text:

1- **Anchors (^ $)**

- ^The matches any string that starts with "The"
- end$ matches any string that ends with "end"
- ^The end$ exact string matches with the text "The end"

2- **Quantifiers (* + ? {})**

- abc* matches any string that has "ab" followed by zero or more "c"
- abc+ matches any string that has "ab" followed by one or more "c"
- abc? Matches any string that has "ab" followed by zero or one "c"
- abc{2} matches any string that has "ab" followed by 2 "c"
- abc{2,} matches any string that has "ab" followed by 2 or more "c"
- abc{2,5} matches any string that has "ab" followed by 2 up to 5 "c"
- a(bc)* matches any string that has "a" followed by zero or more "bc"
- a(bc){2,5} matches any string that has "a" followed by 2 up to 5 "bc"

3- **operators (| [])**

- a(b|c) matches any string that has "a" followed by "b" or "c"

4- **class characters (\d \w \s .)**

- \d match a single character that is a digit
- \w match a word character (alphanumeric i.e. digit or alphabet) also matches _
- \s matches a whitespace character (including tabs and line breaks)
- . matches anything

Methodology

The application is built with python and the GUI library PyQT5. It has three main components:

1- GUI part
2- Spelling checker
3- Normal and regex searcher

Spell checker
Regex search

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern.

Python has a built-in package called **re**, which can be used to work with Regular Expressions.

Results

Spell checker

Firstly, we will initialize the spell checker with the dataset

```
spell_checker = SpellChecker('data/big.txt')
```

Consider a short word like “test”. Let’s say the user typed it as “tdst”.

```
word = 'tdst'
```

Then, we’ll find all these word variations that have edit distance =1. They are 234 variations, lets print the first 50 of these variations.

```
variants1 = spell_checker.variants1(word)
print('number of edit 1 distance variants is {} \n they are {} \n\n'.forma
t(len(variants1), list(variants1)[:50]))
```

```
number of edit 1 distance variants is 234
 they are ['tdstx', 'zdst', 'tqdst', 'tyst', 'tdsht', 'tudst', 'tdsr',
'ptdst', 'tdstb', 'htdst', 'tdsts', 'wtdst', 'ldst', 'tsdst', 'tdse',
'ztdst', 'tnst', 'tdtst', 'tdspt', 'tdsjt', 'tast', 'tdsw', 'tdsxt',
'tbdst', 'tdsth', 'tdstp', 'tdlst', 'tdste', 'qtdst', 'xdst', 'pdst',
'tdzt', 'tqst', 'tddt', 'tdsst', 'tedst', 'tdsot', 'tds', 'tdht', 'tdsdt',
```

```
'tust', 'tdsty', 'tost', 'ntdst', 'tdmst', 'tdyst', 'ttst', 'tdvt',
'tdast', 'xtdst']
```

we'll find all these word variations that have edit distance =2. They are 24253 variations, lets print the first 50 of these variations.

```
variants2 = spell_checker.variants2(word)
print('number of edit 1 distance variants is {} \n they are {} \n\n'.forma
t(len(variants2), list(variants2)[:50]))
```

number of edit 1 distance variants is 24253

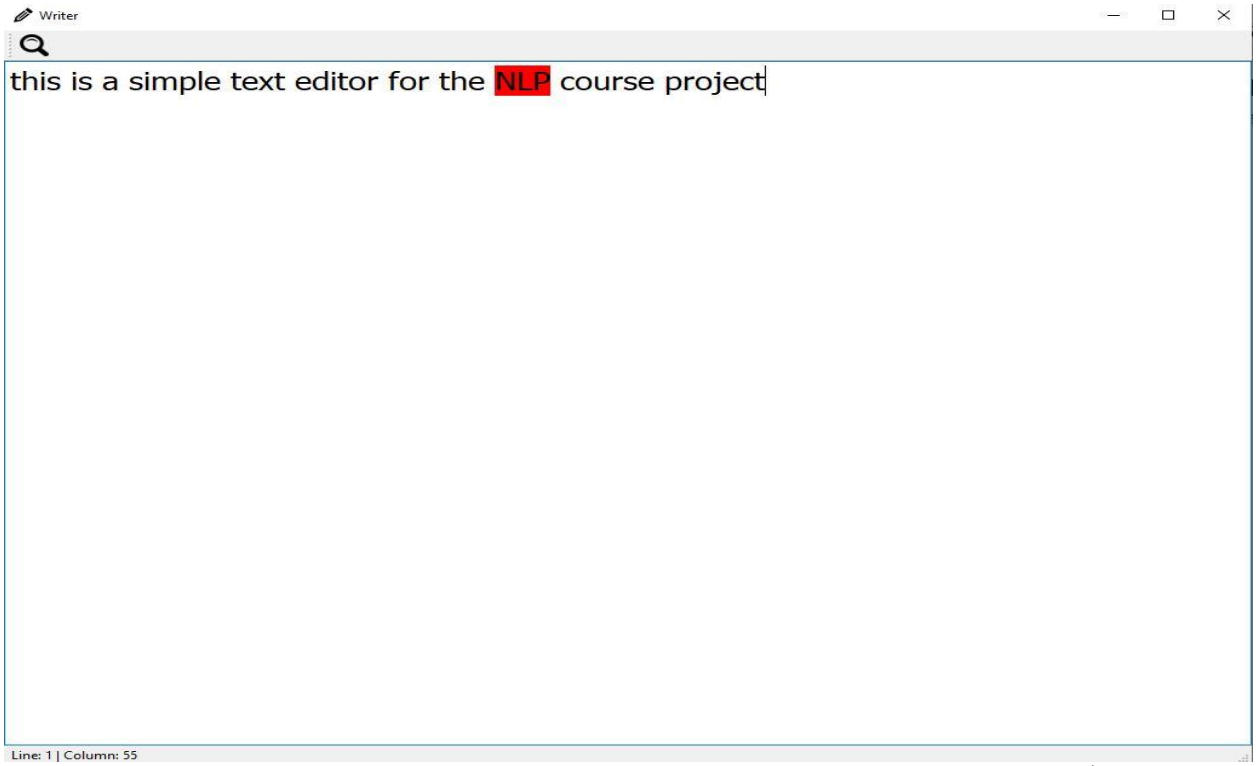
they are ['tdasi', 'ktdsto', 'tmdist', 'lzst', 'tdlsn', 'txdyst', 'tdabt', 'tkdsy', 'tdsmjt', 'wdmst', 'sdsl', 'tdsltj', 'pstdst', 'tpsta', 'gdtst', 'ttst', 'tddsw', 'tdtsmt', 'tbsat', 'tdsqxt', 'tdzst', 'tdsn', 'eltdst', 'tdrstf', 'tdgmt', 'tkdsu', 'tdapst', 'tdsjtx', 'edstn', 'latdst', 'tvdest', 'vdstd', 'tdzpst', 'tnzt', 'todsl', 'tcdyst', 'tzdstb', 'tzdspt', 'ttdsgt', 'tdrtn', 'wtdnt', 'adstn', 'zjdst', 'tdsoa', 'tkdot', 'bdpt', 'tldstz', 'tfdsx', 'tdsdtc', 'zdsu']

After deleting all words that doesn't exist in the corpus, we end up with 1 word only

```
flag, suggesions = spell_checker.check(word)
print("number of suggesions {}, they are {}".format(len(suggesions), sugge
sions))
```

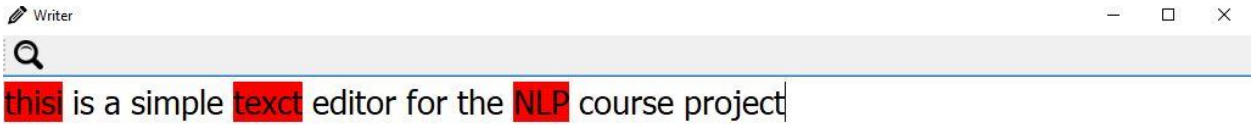
number of suggesions 1, they are {'test'}

The text editor application is shown in the below image

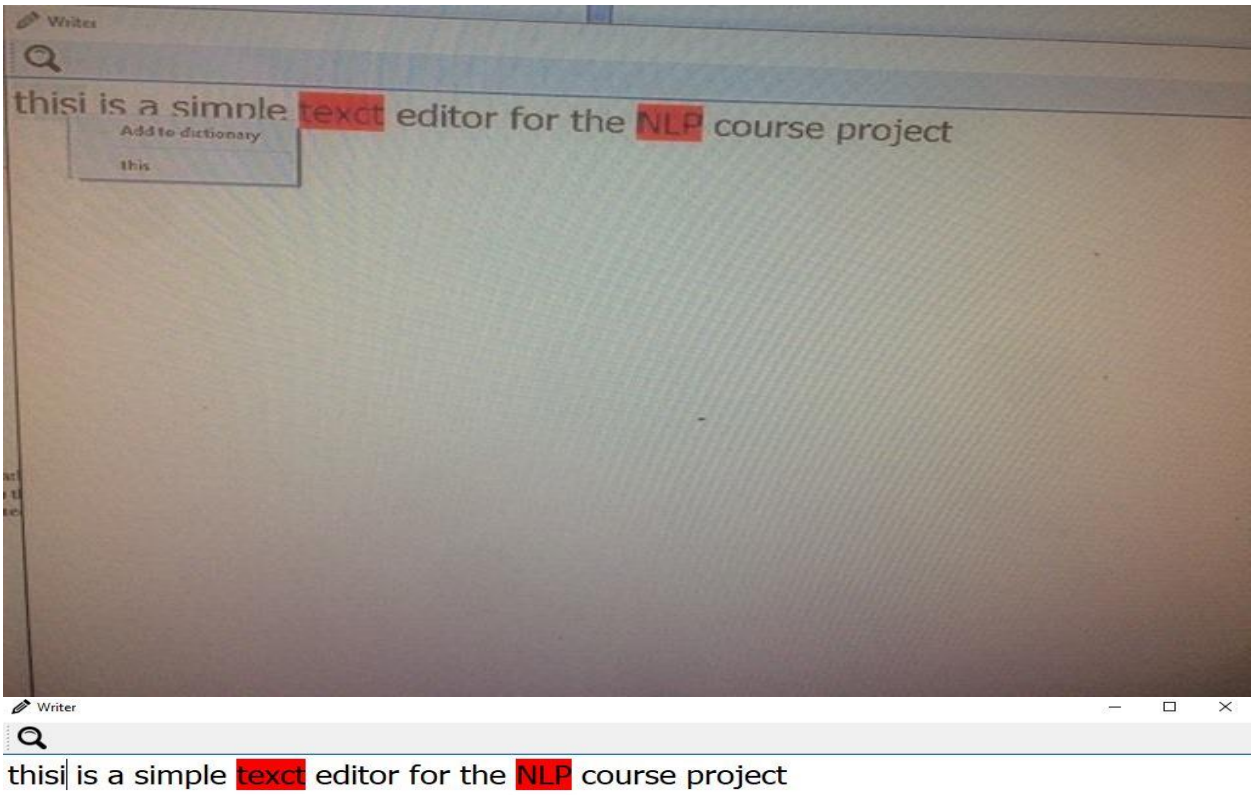


The word NLP is marked as red in the text editor by the application because NLP doesn't exist in the corpus, so the text editor considers it as a wrong word. Also, if we checked the spell checker suggestions to replace NLP, he has none because there is not word of edit distance 1 or 2 to NLP that is exist in the corpus. However, this is because the corpus contains no acronym or may be general ones like USA, that can exist in a novel like The Adventures of Sherlock Holmes.

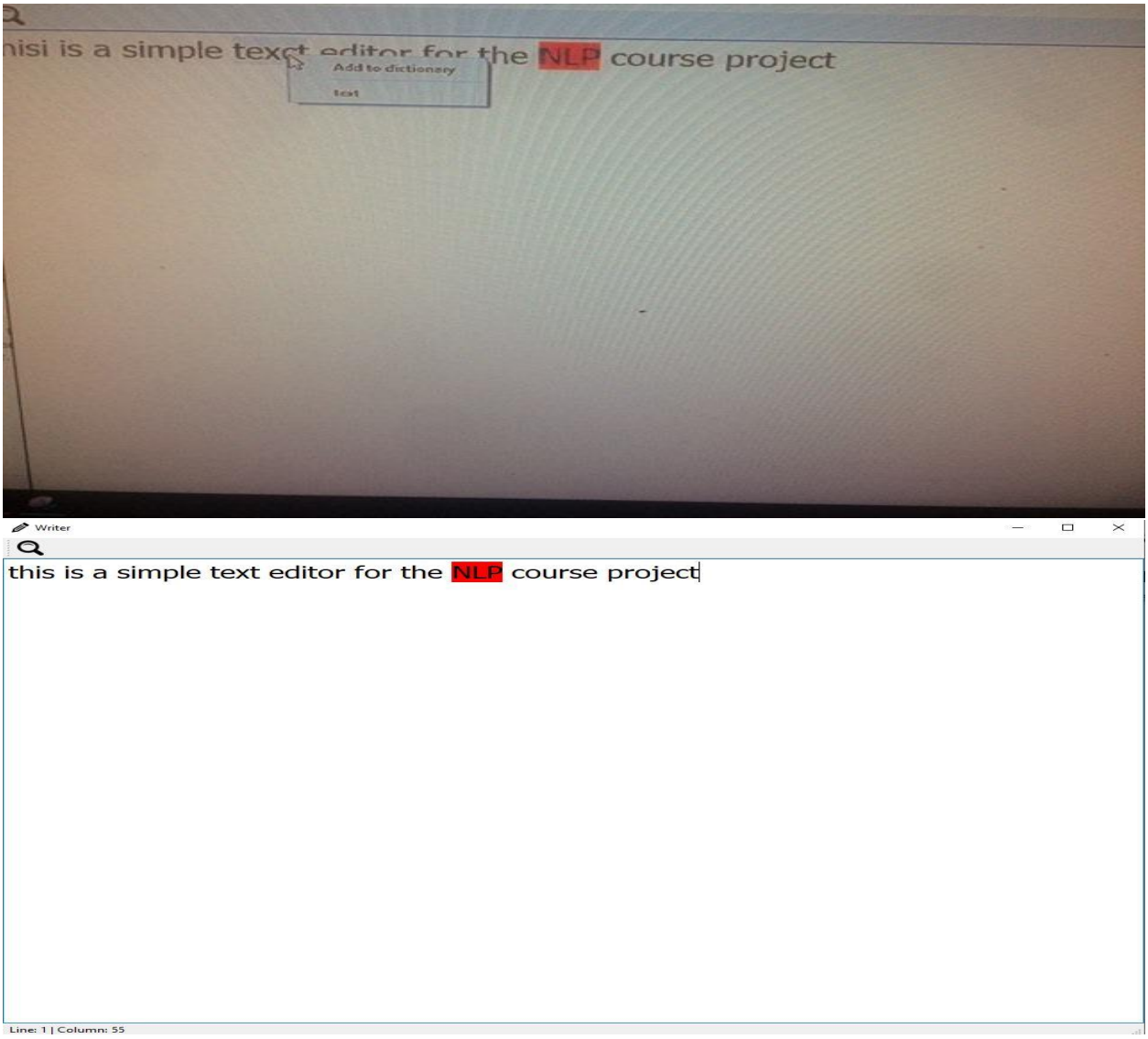
In the next image, we write a similar text but with more non word spelling errors.



As we see ‘thisi’, ‘texct’, and ‘NLP’ are marked by the spellchecker as wrong words. If we right clicked on any of them, we will get a menu that has an ‘add to dictionary’ button that will add this word to the list of right words and the red highlighting will be removed. The next two images show this process.

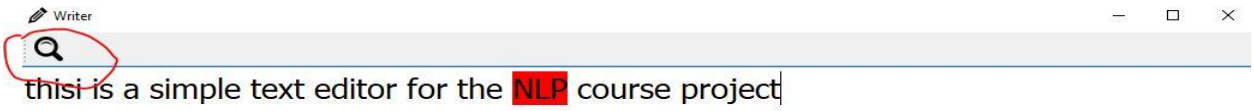


In addition to the ‘add to dictionary’ button, a number of suggestions are listed as buttons where the user can select any of them and the application will replace the word with the selected right word, just as we do in Word processor. This process is shown in the next two images.

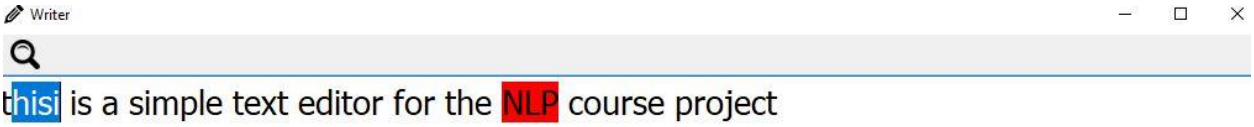


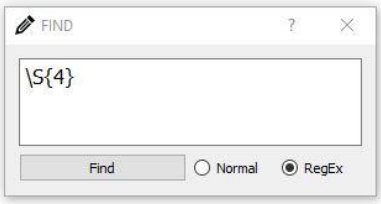
Regex search

In the top left corner there's a search icon which the user can use to perform normal search or regex search. This is shown in the following image.



In the following image, an example of a regex search and the result selected pattern is shown.





Discussion and conclusion

In this report, two of the most useful and widely used NLP features were explored by integrating them in a text editor application. The first is spell checking and correction where the application is first fed by a corpus of correct word. Each word the user inputs is then compared against this corpus and if it's not found it's flagged as a wrong word then the user is presented with suggestions to replace this word or the option to add it to the dictionary. Secondly, regular expression pattern matching is integrated along with normal search. Regex is a powerful information extraction technique as it allows the user to search for patterns that have specific format without knowing exactly the data this pattern has. It can be use to serach for dates, phone numbers, money amounts, temperature degrees, or any other kind of data that follow a standard format.

Appendix

1- Function that generates the language model from the input text and tokenized its words.

```
    def get_language_model(self, file_name):

        language_model = collections.defaultdict(lambda: 0)

        with open(file_name) as file :
            txt = file.read()

        words =  re.findall('\w+', txt.lower())

        for word in words:
            language_model[word] += 1

        real_words = set(language_model)
        return language_model, real_words
```

2- Function that generates possible candidates that are at edit distance of the wrong word.

```
    def variants1(self, word):
        """get all possible variants for a word"""
        splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
        deletes    = [a + b[1:] for a, b in splits if b]
        transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]
        replaces   = [a + c + b[1:] for a, b in splits for c in self.alphabet if
b]
        inserts    = [a + c + b for a, b in splits for c in self.alphabet]
        return set(deletes + transposes + replaces + inserts)
```

3- Prioritize suggestions to user as per the error model discussed earlier in this report.

```
    def suggestions(self, word ) :
        return self.variants1(word) & self.real_words or \
               self.variants2(word) & self.real_words or \
               {word}

    def check(self, word) :
        if word in self.real_words :
            return True, word

        return False, self.suggestions(word)
```

References

1. . Peter Norvig. (2020). Retrieved 3 June 2020, from http://norvig.com/
2. How to Write a Spelling Corrector. (2020). Retrieved 3 June 2020, from http://norvig.com/spell-correct.html
3. Searching with Regular Expressions (RegEx). (2020). Retrieved 3 June 2020, from https://help.relativity.com/9.5/Content/Relativity/Regular_expressions/Searching_with_regular_expressions.html
4. gadm21/SimpleTextEditor. (2020). Retrieved 3 June 2020, from https://github.com/gadm21/SimpleTextEditor