Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO

TRABALHO PRÁTICO 2

Fuzzywuzzy – Fuzzy String Matching

Autores:

Bruno Rodrigues: pg41066

Carlos Alves pg41840

1 ÍNDICE

2 INTRODUÇÃO

Este trabalho tem como objetivo abordar e estudar a biblioteca **FuzzyWuzzy**. Foi-nos proposto na unidade curricular Introdução ao Processamento de Linguagem Natural que realiza-se-mos o estudo da ferramenta/biblioteca **FuzzyWuzy** ou mais conhecida por *fuzzy string matching*.

Em conformidade com o que foi mencionado acima, este documento foi redigido de modo a apresentar a biblioteca **FuzzyWuzzy** e as suas funcionalidades. Inicialmente, abordaremos o tema *Distância de Levenshtein*, devido a este ser um tema abordado pela biblioteca em estudo. Também constará neste documento pequenos exercícios, utilizando a ferramenta de forma a ser possível uma melhor compreensão sobre a mesma. Por fim seremos capazes de realizar um exercício mais completo no contexto de Processamento de Linguagem Natural.

3 FUZZYWUZZY

3.1 OBJETIVO

FuzzyWuzzy é uma biblioteca de Python que tem como objetivo ser usada para correspondências de *strings*. **Fuzzy string matching**, nada mais é que o processo de encontrar *strings* que correspondem a um determinado padrão, aproximadamente. Basicamente o **FuzzyWuzzy** faz uso de *Levenshtein Distance*, para calcular as diferenças entre as *strings*.

FuzzyWuzzy foi desenvolvido pela *SeatGeek*, um serviço para encontrar bilhetes para festivais e jogos de diversas modalidades.

Antes de abordar em concreto a biblioteca, achamos por bem contextualizar a *Distância Levenshtein*, visto que a biblioteca em estudo faz uso continuo dessa mesmo Distância.

3.2 DISTANCIA LEVENSHTEIN

O russo Vladimir Levenshtein, em 1965 considerou esta distância sendo muito útil para aplicações que precisam determinar quão semelhantes duas *strings* são. Exemplo: Concelho e conselho.

Sucintamente, a *Distância de Levenshtein* (ou edição) entre duas strings é dada pelo número mínimo de operações necessárias para transformar uma *string* noutra. Aqui podem ser efetuadas certas operações como: inserção, remoção ou até mesmo substituição de um caracter.

O algoritmo que é mais utilizado para calcular esta distancia, usa uma matriz (n+1)*(m*1), onde n e m são o numero de carateres das duas *strings*.

Abaixo é possível observar um pseudocódigo retirado de *rosettacode*.

```
//Pseudocodigo funçao DistanciaLevenshtein
Função DistanciaLevenshtein(Caracter : str1[1..lenStr1], Caracter : str2[1..lenStr2]) : INTEIRO
Início
   // tab é uma tabela com lenStr1+1 linhas e lenStr2+1 colunas
   Inteiro:  tab[0..lenStr1, 0..lenStr2]
   // X e Y são usados para iterar str1 e str2
   Inteiro:  X, Y, cost

   Para X de 0 até lenStr1
       tab[X, 0] ← X
   Para Y de 0 até lenStr2
       tab[0, Y] ← Y

   Para X de 1 até lenStr1
       Para Y de 1 até lenStr2
           Se str1[X] = str2[Y] Então cost ← 0
                                Se-Não cost ← 2 // Custo da substituição deve ser 2, remoção e
inserção
           tab[X, Y] := menor(
                                tab[X-1, Y  ] + 1,     // Remover
                                tab[X  , Y-1] + 1,     // Inserir
                                tab[X-1, Y-1] + cost   // Substituir
                           )
   DistanciaLevenshtein ← tab[lenStr1, lenStr2]
 Fim
```

Figura. 1 – Pseudocódigo função **Distância de Levenshtein**

Onde **srt1[1..lenStr1]** e **srt2[1..lenStr2]** são sequencias de caracteres que podem ser substituídas uma pela outra, usando o minimio de operações possíveis **tab[X,Y]**. Por fim, o algoritmo termina com o elemento lado direito do array que contém a resposta.

Ao analisar este pseudocódigo concluímos que é possível ainda melhorá-lo, pois o algoritmo peca em alguns aspetos como: na paralelização (muitas dependências de dados), no armazenamento do número de inserções, remoções e substituições, entre outras. Desde logo, podíamos dar diferentes "penalidades" de custo às inserções, substituições e remoções. No quesito da paralelização podíamos fazer com que todo o custo fosse calculado em paralelo e ainda fazer uma função mínimo em fases, deste modo eliminaríamos as dependências. Existem diversas melhorias que poderiam ser realizadas neste algoritmo.

Anteriormente foi referido que o segmento inicial **str1[1..X]** pode ser transformada no segmento **str2[1..Y]** usando um mínimo de **tab[X,Y]** operações, a isto chamamos de constante. Esta constante verifica-se verdadeira: pois é inicialmente verdadeira na linha e coluna 0, porque **srt1[1..X]** pode ser transformada numa *string* vazia, isto é, **str2[1..0].** Bastando apenas apagar todos os X carateres. É possível fazer o inverso, onde podemos transformar **str1[1..0]** em **str2[1..Y]** adicionando todos os carateres Y. Outra prova é que as operações necessárias para transformar **str1[1..n]** em **str2 [1..m]** é basicamente o número necessário para transformar todos os str1 em todos os str2, e imediatamente o **tab[n,m]** contem o resultado desejado.

Contudo com esta prova não é possível confirmar que o número colocado em **tab[X,Y]** seja realmente o mínimo.

Em relação aos limites, a *distância Levenshtein* apresenta diversos limites superiores e inferiores que são úteis em aplicações que calculam vários deles e ainda os compara. Abaixo temos alguns desses limites:

- É pelo menos a diferença dos tamanhos das duas sequências.
- É no máximo o comprimento da *string* mais longa.
- É zero se e somente se as sequências de caracteres forem iguais.
- No caso das *strings* tiverem o mesmo tamanho, a *distância de Hamming* será um limite superior da *distância de Levenshtein*.

Para concluir e para melhor compreensão sobre a *distância de Levenshtein* usada pela biblioteca abordada, **FuzzyWuzzy**, elaboramos um exemplo simples abaixo.

Se tivermos a palavra “CORASSÃO” (palavra com erros ortográficos) a *distância de Levenshtein* ajudar-nos-á a sugerir a palavra “CORAÇÃO” (palavra com a grafia correta) como uma correção, visto que o valor da distancia de edição entre as duas palavras é igual a 2 - valor baixo.

C	O	R	A	S	S	Ã	O

C	O	R	A	Ç	Ã	O	

```
'''Remoção'''
string2 = "Corassão"
string2 = string2[:4] + string2[5:]
print(string2)#Corasão

'''Substituição'''
string3 = string2
string3 = string3[:4] + "ç" + string3[5:]
print(string3)#Coração
```

Figura. 2 – Remoção e Substituição

Basta realizar 1 remoção na palavra “CORASSÃO” e 1 substituição para obtermos a palavra correta, “CORAÇÃO”.

```python
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0
    helper.__name__= func.__name__
    return helper

dicmemo = {}#dicionario memoria

@call_counter
def levenshtein(s, t):
    if s == "":
        return len(t)
    if t == "":
        return len(s)
    cost = 0 if s[-1] == t[-1] else 1

    i1 = (s[:-1], t)
    if not i1 in dicmemo:
        dicmemo[i1] = levenshtein(*i1)
    i2 = (s, t[:-1])
    if not i2 in dicmemo:
        dicmemo[i2] = levenshtein(*i2)
    i3 = (s[:-1], t[:-1])
    if not i3 in dicmemo:
        dicmemo[i3] = levenshtein(*i3)
    res = min([dicmemo[i1]+1, dicmemo[i2]+1, dicmemo[i3]+cost])

    return res
print(levenshtein("Corassão", "Coração"))
print("A função foi chamada " + str(levenshtein.calls) + " vezes!")

```

Figura. 3 – Exemplo do algoritmo descrito acima.

```
2
A função foi chamada 72 vezes!
```

Figura. 4 – Output

Por fim a *Distância de Levensthein* tem inúmeras utilizações tais como:

- Correspondência aproximada de *strings*, onde o objetivo é encontrar correspondências para *strings* curtas em vários textos de tamanho arbitrário.
- *Spell checkers*;
- Sistemas de correção para reconhecimento ótico de carateres;
- E ainda aplicações que ajudam na tradução de Linguagem Natural com base na memória de tradução (banco de dados que armazena “strings”).

3.3 FUZZYWUZZY

Voltando para a biblioteca em estudo, **FuzzyWuzzy**, como já referido, é uma biblioteca de Python usada para correspondência de *strings*. Faz uso da *Distância de Levenshtein* (descrito anteriormente) para calcular as diferenças entre as *stings*.

Ao fazer uso de string matching, a correspondência de fuzzy string matching é um tipo de pesquisa que encontrará correspondências mesmo quando os usuários digitam incorretamente as palavras ou até mesmo quando os utilizadores digitam "meias palavras" de modo a ser completada.

3.3.1 ESTRUTURA

- **StringMatcher**: Classe importada da biblioteca **python-Levenshtein**, esta classe é semelhante ao **SequenceMatcher** criada dessa biblioteca "externa". Parecido com o que foi explicado no início deste documento (*Distancia de Levenshtein*/ Distancia de edição).
- **Fuzzy**: Classe que implementa a verificação de ratio entre *strings* de diferentes formas. Funções serão descritas posteriormente.
- **Process**: Responsável por encontrar as melhores correspondências numa lista ou dicionários, devolve um gerador de tuplas que contem a correspondências e os scores, isto com base nos limites. No caso de ser usado dicionário, também será devolvido uma "chave" para cada correspondência. Possui métodos que devolvem apenas a melhor correspondência possível e ainda faz uso do *fuzzy matching*.
- **String processing**: Classe contém métodos que processam *strings* com a melhor eficiência possível. Todas as funções usam **unicode** *strings* tanto para o input como para o output. Única classe da biblioteca **fuzzywuzzy** que usa **regex's** para substituir strings que não possuem letras e números com um único espaço em branco.
- **Utils**: Contém métodos variados, em geral destinados para a verificação de strings (se são iguais, se são "vazias" e ainda o comprimento das mesmas).

Em suma, a biblioteca é bastante compacta e reduzida em termos de código.

3.3.2 FUNCIONALIDADES

Pode ser usado em aplicações de correspondência aproximada, incluindo verificação e correção ortográfica. Por exemplo, ao pesquisar no google algo como “arroz de cabide” sendo a que a frase correta seria “arroz de cabidela”, a pesquisa irá devolver resultados mesmo que o input contenha carateres em falta ou a mais, ou até mesmo erros ortográficos, sugerindo ainda a correção. O exemplo dado pode ser verificado na imagem abaixo.

Figura. 5 – Correspondência aproximada

Fuzzy string matching pode também ser usada na filtragem de spam e na vinculação de registos, isto é, uma aplicação comum onde os registos de 2 bancos de dados diferentes são correspondidos.

3.3.3 EXEMPLOS DE UTILIZAÇÃO

Existem diversas formas de comparar duas *strings* com a biblioteca **Fuzzywuzzy**. Uma delas é o uso do *ratio,* que compara a similaridade das *strings*, ordenadamente.

```
1  #!/usr/bin/env python3
2  from fuzzywuzzy import fuzz
3  from fuzzywuzzy import process
4  #Compara o quão similares são as strings, por ordem
5  print(fuzz.ratio('Pica no chão','Pica na mesa'))
```

Figura. 6 – Ratio entre 2 strings

“Pica no chão" e "Pica na mesa" tem uma similaridade de 58%.

```python
#!/usr/bin/env python3
from fuzzywuzzy import fuzz
from fuzzywuzzy import process
#Compara o quão similares são as strings, por ordem
print(fuzz.ratio('reveillon','reiveilon'))
```

Figura. 7 – Ratio entre 2 *strings*

Caso as *strings* sejam: “reveillon” e “reiveilon”, obteremos uma similaridade de 89%.

Usando o *ratio*, concluímos que este tipo de abordagem é demasiado sensível a pequenas discrepâncias nas *strings*, isto é, carateres em falta ou a mais.

Uma outra forma de comparar *strings*, é usando o *partial_ratio,* que compara a similaridade parcial das *strings*.

De seguida temos o *token_sort_ratio* que ignora a ordem das palavras. Verificamos que em sequências de palavras com mais que 2 palavras obtemos uma maior percentagem de similaridade. Nas *strings* de apenas 1 palavra este token_sort_ratio não fornece melhores resultados.

E por último temos *token_set_ratio* que ignora palavras duplicadas e ainda ignora a ordem das palavras, tornando-se na operação mais flexível entre todas as que apresentamos acima. Além disso *WRatio* também é uma boa operação a se ter em conta, pois o *WRatio* consegue lidar com maiúsculas e minúsculas entre outros parâmetros (o mais completo), e ainda existem mais 3 métodos: *Qratio*, *UQratio*, *UWRatio*(semelhantes ás referidas acima).

```python
#!/usr/bin/env python3
from fuzzywuzzy import fuzz
from fuzzywuzzy import process

str1 = 'Pica no chão com com muito vinagre'
str2 = 'A galinha pica no chão, mas leva com o vinagre'

print ('Ratio: ', fuzz.ratio(str1, str2))
print ("PartialRatio: ", fuzz.partial_ratio(str1, str2))
print ("TokenSortRatio: ", fuzz.token_sort_ratio(str1, str2))
print ("TokenSetRatio: ", fuzz.token_set_ratio(str1, str2))
print ("WRatio: ", fuzz.WRatio(str1, str2),'\n\n')

# Usando a biblioteca process
query = 'pudins de laranja'
choices = ['pudim de laranja', 'pudim pudim', 'laranja de pudim']
print ("Lista de ratios: ", process.extract(query, choices), '\n')
print ("Melhor da lista: ", process.extractOne(query, choices))

```

Figura. 8 – Utilizando a biblioteca **FuzzyWuzzy**

Na figura acima temos uma demonstração de como fazer uso das diferentes componentes da biblioteca Fuzzywuzzy. Uma delas, o **process**, que nos permite encontrar a correspondência mais próxima da *string* colocada inicialmente.

No **process** temos:

dedupe() – Função que pega numa lista de *strings* com palavras duplicadas e usa fuzzy matching para identificar e remover essas palavras duplicadas. Primeiro, usa o *process.extract* para identificar tais palavras na lista, conforme o score definido pelo utilizador. Depois procura o item mais longo na lista de palavras duplicadas, pois é suposto esse item contenha mais informações sobre entidade e devolve.

```
Returns:
    A deduplicated list. For example:

        In: contains_dupes = ['Frodo Baggin', 'Frodo Baggins', 'F. Baggins', 'Samwise G.', 'Gandalf', 'Bilbo Baggins']
        In: fuzzy_dedupe(contains_dupes)
        Out: ['Frodo Baggins', 'Samwise G.', 'Bilbo Baggins', 'Gandalf']
```

Figura. 9 – Exemplo **dedupe**()

extract() – Escolhe a melhor correspondência numa lista ou dicionário de opções. Devolve uma lista de tuplas contendo a correspondência e o seu respetivo score. Caso seja usado um dicionário, também será devolvido a chave para cada correspondência.

```
Returns:
    List of tuples containing the match and its score.

    If a list is used for choices, then the result will be 2-tuples.
    If a dictionary is used, then the result will be 3-tuples containing
    the key for each match.

    For example, searching for 'bird' in the dictionary

    {'bard': 'train', 'dog': 'man'}

    may return

    [('train', 22, 'bard'), ('man', 0, 'dog')]
```

Figura. 10 – Exemplo extract().

extractBests() – Devolve uma lista das melhores correspondências para uma coleção de opções.

extractWithoutOrder() – Semelhante à função **extract**().

extractOne() – Encontra a melhor correspondência única acima de uma pontuação numa lista.

3.4 EXERCÍCIO NO CONTEXTO DE PNL

Como referido em cima, a biblioteca **FuzzyWuzzy** pode ser bastante útil na criação de *spellcheckers*, algo que este exercício se vai focar.

Para tal, desenvolvemos um código para demostrar a capacidade da biblioteca para estes casos.

Em baixo encontra-se um exemplo de um *spellchecker* simples:

```
def ill_Correct(corrString, word_List):

    palavrinhas = corrString.split()
    print(palavrinhas)
    newfrase = []
    # loop das palavras a serem verificadas
    for i in range(len(palavrinhas)):

        max_ratio = 0
        newstring = ''
        # verifica na lista de palavras
        for name in GetWords(word_List):
            # calcula o ratio da palavra
            ratio = fuzz.ratio(palavrinhas[i], name)
            # se for 100, ou igual, acaba aqui
            if ratio == 100:
                newstring = name
                print(newstring + " " + str(ratio) + "%")
                break
            # se for 60
            if ratio >= 60:
             #e maior que a anterior
                if ratio > max_ratio:
                    # vai ficar com ela
                    newstring = name
                    print(newstring + " " + str(ratio) + "%")
                    max_ratio = ratio
        print(max_ratio)
        newfrase.append(newstring)

    return " ".join(newfrase)
```

Figura. 11 – Excerto programa desenvolvido.

O código funciona da seguinte forma:

O utilizador escreve uma frase, preferencialmente com erros ortográficos, e o que a função vai fazer é, pegar em cada palavra que constitui a frase, e verificar se existe correspondência a partir de uma lista de palavras providenciada. É possível utilizar para corrigir ficheiros texto, mas o tempo necessário para corrigir o texto pode ser demoroso.

```
#texto com as palavras
with_this_word_List = 'wordlist-preao-20190329.txt'
print('Escrebe mal portuges')
this_String = input()
print(ill_Correct(this_String,with_this_word_List))
```

Figura. 12

Caso a palavra exista, esta não será modificada, e então manter-se-á na *string* final.

Caso a palavra não exista, o programa vai tentar encontrar a palavra com melhor correspondência, e substitui-a.

Neste caso em particular, utilizamos apenas o método **fuzz.ratio** para verificação de correspondência entre palavras, sendo este método mais rápido do que outros mais completos, como **fuzz.Wratio**, mas para obter melhores resultados, seria necessário utilizar outros métodos da biblioteca FuzzyWuzzy em conjunto com outras bibliotecas, mas como o objetivo deste exercício é apenas demonstrar as potencialidades da biblioteca em estudo, achamos que seria irrelevante o uso de outras bibliotecas.

Em baixo segue-se um exemplo de output do código:

```
bruno@bruno-Aspire-E5-575G:~/Documentos/EXIPLN$ python3 fuzzy.py
Escrebe mal portuges:
eu sei escreber otorinolarigolaringo
['eu', 'sei', 'escreber', 'otorinolarigolaringo']
Deus 67%
deu 80%
eu 100%
80
Hussein 60%
abusei 67%
asei 86%
sei 100%
86
Heisenberg 67%
adscrever 71%
descaber 75%
descrer 80%
descrever 82%
escrever 88%
88
otorrinolaringologia 75%
75
eu sei escrever otorrinolaringologia
```

Figura. 13 – Output do exemplo desenvolvido

Como podemos ver, a *string* é dividida por palavras, depois cada palavra é verificada na lista de palavras disponibilizada.

Caso este encontre uma palavra com 100% de correspondência, esta é devolvida, se não, irá ser retornada a palavra com maior correspondência, como é o caso das palavras “escrever” e “otorrinolaringologia”.

Neste caso, a palavra que pretendíamos obter era “otorrinolaringologista”.

O exemplo encima demonstra o potencial da biblioteca para usos de *spellcheck*, mas para obter um *spellchecker* mais viável, teria de existir um refinamento maior do código, e aplicar outros métodos de diferentes bibliotecas em conjunto com a biblioteca **FuzzyWizzy**, para uma precisão maior.

4 CONCLUSÃO

Com o desenvolvimento deste trabalho, descobrimos que a ferramenta **FuzzyWuzzy/Fuzzy String Matching** é uma ferramenta poderosíssima e bastante simples de se utilizar, visto que não tem um número de funções "absurdo". Além disso, a pesquisa realizada para descrever o seu funcionamento, "transportou-nos" para um outro tema, *Distância Levensthein* ou distância de edição, biblioteca usada pelo **FuzzyWuzzy**. No quesito desta Distância, concluímos que existem ainda diversos problemas aliados à distância de edição. Pois o algoritmo a ser utilizado deve sempre ser escolhido, conforme as necessidades da aplicação que se deseja desenvolver, e nunca utilizar os comuns e mais usados algoritmos.

Em relação à própria biblioteca **FuzzyWuzzy**, concluímos que esta pode ser facilmente adaptada a aplicações diversas. Tanto para a correspondência de nomes de clientes como para a correção ortográfica. Pode economizar muitos recursos, mas necessita de pré-processamento e ainda não tem em conta sinónimos. Deste modo, facilmente concluímos que esta pode não ser a melhor escolha a ser usada para casos que sejam necessárias técnicas de Processamento de Linguagem Natural, por exemplo incorporar palavras como o algoritmo Clustering.

5 REFERÊNCIAS

(s.d.). Obtido de pyhton-course: https://www.python-course.eu/levenshtein_distance.php

Gonzalo Navarro, R. B.-Y. (2001). *Indexing Methods for Approximate String Matching.* IEEE Data Engineering Bulletin.

SeatGeek. (2015). *github.* Obtido de SeatGeek: https://github.com/seatgeek/fuzzywuzzy

Sharma, S. (2018). *Geeksforgeeks*. Obtido de https://www.geeksforgeeks.org/fuzzywuzzy-python-library/

Wikipedia. (s.d.). Obtido de Wikipedia: https://en.wikipedia.org/wiki/Approximate_string_matching

Lista de Palavras utilizada no spellchecker:

http://natura.di.uminho.pt/download/sources/Dictionaries/wordlists/