Mesure	Exemple
La Distance de Levenshtein : Elle retourne un nombre, de suppressions, insertions, substitutions de caractères nécessaires pour transformer l'un des chaines en l'autre.	***k**itt**e**n et **s**itt**i**n**g** sont à distance 3 : **k**itten → **s**itten (substitution de "s" à "k") sitt**e**n → sitt**i**n (substitution de "i" à "e") sittin → sittin**g** (insertion de "g" à la fin).
L'Identité : Le résultat est de zéro pour l'absence de similarité et d'un pour l'égalité des chaines comparées.	S1 = S2 ➔ 1 S1 /= S2 ➔ 0
L'indice de Jaccard : Consiste à diviser le nombre d'objets en commun par le nombre d'objets distincts dans les deux ensembles. Inconvénient : une correspondance parfaite entre les chaines est exigée.	Le cardinal de l'intersection par/ le cardinal de l'union (U) ou, de façon équivalente, en divisant la différence de la taille de l'union et de l'intersection des deux ensembles par la taille de l'union.
La Distance de Jaro-Winkler : indique le score de similarité entre deux chaines. La mesure Jaro est le nombre minimum de transpositions d'un caractère requis pour changer un mot dans l'autre.Winkler a augmenté cette mesure pour les caractères initiaux correspondants (les préfixes). Ce qui donne des notes plus favorables aux chaînes qui correspondent depuis le début pour une longueur de préfixe fixe.	Dj = (m/s1 + m/s2 + m-t/m) m = nombre de caractère qui match t = est la moitié du nombre de transpositions s = Chaine de caractère
Le Coefficient de chevauchement : (ou coefficient de Szymkiewicz-Simpson) Est une mesure de similarité liée à l'indice de Jaccard qui mesure le chevauchement entre deux ensembles, et est définie comme la taille de l'intersection divisée par la plus petite des deux ensembles.	En appliquant cette méthode sur deux chaines X et Y. Si l'ensemble X est un sous - ensemble de Y ou l'inverse, alors le coefficient de recouvrement est égal à un.
La distance de Hamming : entre deux chaînes de longueur **égale** est le nombre de positions auxquelles les symboles correspondants sont différents. En d'autres termes, il mesure le nombre minimum de substitutions nécessaires pour changer une chaîne dans l'autre, ou le nombre minimum d'erreurs qui	Entre "ka rol in" et "ka thr in" est 3. "k un r ol dans" et "k e r st dans" est 3. « 10 1 1 1 01 » et « 10 0 1 0 01 » est 2. « 2 17 3 8 96 » et « 2 23 3 7 96 » est 3.

auraient pu transformer une chaîne dans l'autre.	

Une implémentation de **la méthode de Jaro** :

```
if len (m1) ==  0  ou  len (m2) ==  0
      return  0.0
else len (m1) ==  len (m2):
    Return  0.0
 return ( float ( len (m1)) /  len (plus court) +
         float ( len (m2)) /  len (plus long) +
        float ( len (m1) - _transpositions (m1, m2)) /  len (m1)) /  3.0
```

Quelques exemples de **la distance de Levenstein** :

1) Chaines de caractères simples :

Prenons les deux chaines suivantes : Bon, Pont

*La méthode de remplacement :

```
s = "Bon"
s = s[:0] + "P" + s[0:]     #la position 0 ici est la lettre B puis P après remplacement.
print (s)
 'Pon'
```

*La méthode d'ajout :

```
s = "Pon"
s = s[:3] + "t"   #la position 3 ici est vide et viens après la lettre « n »
 print(s)
  Pont
  #La distance de Levenstein ici est 2 : c'est le nombre minimal de remplacements ou de modification.
```

La méthode Levenstein calcule le nombre minimal de modification entre deux chaines de caractères.

2) Chaines décomposées en sous-chaines :

Prenons les chaines : Maison – Raison qu'on va décomposer de la sorte :
t1 = ("ma", "is", "on")
t2 = ("ra", "is", "on")

Donc distance.levenshtein(t1, t2) est de 1. Remplacement de « ma » par « ra ».

3) Chaines Complexes :

ch1 = ['je', 'suis', 'etudiant', 'en', 'master','informatique','pour','science']

ch2 = ['je', 'suis', 'etudiant', 'a', 'la fac', 'informatique']

>>> distance.levenshtein(sent1, sent2)

Ici nous avons effectué 4 Opérations : 2 remplacement de la lettre « a » par « en », et de « la fac » est remplacé par « master». Ainsi que 2 Ajouts de « pour » et de « science ».

Exemple sur Hamming :
(https://www.revolvy.com/main/index.php?s=Hamming+distance&item_type=topic)

La fonction « hamming_distance », implémentée en Python 2.3+ , calcule la distance de Hamming entre deux chaînes (ou autres objets itérables) de longueur égale en créant une séquence de valeurs booléennes indiquant des discordances et des correspondances entre les positions correspondantes dans les deux entrées puis en sommant la séquence avec False et les valeurs vraies étant interprétées comme zéro et un.

```
def hamming_distance ( s1 , s2 ):     "" "Retourne la distance de Hamming entre les séquences de longueur égale" ""     si len ( s1 ) =! len ( s2 ):         augmenter ValueError ( "non définie pour les séquences de longueur inégale" )     retour somme ( el1 ! = el2 pour el1 , el2 dans zip ( s1 , s2 ))
```

Où la fonction zip fusionne deux collections de longueur égale par paires.

Implémentation de l'indice de Jaccard en Python :

```
def compute_jaccard_index(set_1, set_2):
    return len(set_1.intersection(set_2)) / float(len(set_1.union(set_2)))


def compute_jaccard_index(set_1, set_2):
    n = len(set_1.intersection(set_2))
    return n / float(len(set_1) + len(set_2) - n)
```

#Vous constaterez que nous n'avons pas vraiment besoin de calculer l'ensemble d'union, mais plutôt la cardinalité.Donc ce code Fonctionne mieux.

Un autre exemple de Jaccard :

(http://bugra.github.io/work/notes/2017-02-07/similarity-via-jaccard-index/)

```
def  jaccard_index ( first_set ,  second_set ):
   "" "Calcule l'index jaccard de deux ensembles
      Arguments:
        first_set (set):
        second_set (set):
      Retourne:
        index (float): Indice Jaccard entre deux ensembles, il est compris entre 0.0 et 1.0
   " " "
   # Si les deux ensembles sont vides, l'indice de Jaccard est défini comme 1
   indice =  1,0
   si  first_set  ou  second_set :
   indice =  ( float ( len ( first_set . intersection (second_set )))
          /  len ( premier_set . union ( second_set )))


    indice de retour


first_set  =  set ( plage ( 10 ))
second_set  =  set ( plage ( 5 ,  20 ))
index  =  jaccard_index ( premier_set ,  second_set )
print ( index )  # 0.25 en 5/20
```

Comparaison des différentes mesures de similarités :

- Levenshtein distance : fournit une mesure de similarité entre deux chaînes. Elle permet la suppression, l'insertion et la substitution.
- Jaro : Fournit une mesure de similarité entre deux chaînes permettant des transpositions de caractères.
- Jaro-Winkler : Fournit d'une mesure de similarité entre deux chaînes permettant des transpositions de caractères à un degré ajustant la pondération pour les préfixes communs.
- La distance de Hamming ne permet que la substitution, par conséquent, elle ne s'applique qu'aux chaines de même longueur.
- Jaccard Index est une statistique pour comparer et mesurer à quel point deux ensembles différents sont similaires.

STOP WORD ou Suppression des mots vides :

(https://www.geeksforgeeks.org/removing-stop-words-nltk-python/)

Stop Words: un mot d'arrêt est un mot courant (tel que "Le", "un", "une", "dans") qu'un moteur de recherche a été programmé pour ignorer, à la fois lors de l'indexation des entrées et lors de la recherche à la suite d'une requête de recherche.

Nous ne voudrions pas que ces mots prennent de la place dans notre travail, ou prennent du temps de traitement précieux. Pour cela, il est possible de les supprimer facilement en stockant une liste de mots considérés comme des mots vides. NLTK (Natural Language Toolkit) en python à une liste de mots vides stockés dans 16 langues différentes. On peut les trouver dans le répertoire nltk_data. home / pratima / nltk_data / corpora / stopwords (est l'adresse du répertoire).

* Pour vérifier la liste des mots vides, vous pouvez taper les commandes suivantes dans le shell python :

```
import nltk

from nltk.corpus import stopwords

 set(stopwords.words('english'))
```

* Suppression des mots d'arrêt avec NLTK

Le programme suivant supprime les mots d'arrêt d'un texte:

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

```
example_sent = "This is a sample sentence, showing off the stop words filtration."
stop_words = set(stopwords.words('english'))
word_tokens = word_tokenize(example_sent)
filtered_sentence = [w for w in word_tokens if not w in stop_words]
filtered_sentence = []
for w in word_tokens:
    if w not in stop_words:
        filtered_sentence.append(w)
print(word_tokens)
print(filtered_sentence)
```

* Effectuer les opérations Stopwords dans un fichier

Dans le code ci-dessous, text.txt est le fichier d'entrée d'origine dans lequel les mots vides doivent être supprimés. filteredtext.txt est le fichier de sortie. Cela peut être fait en utilisant le code suivant:

```
import io
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
#word_tokenize accepts a string as an input, not a file.
stop_words = set(stopwords.words('english'))
file1 = open("text.txt")
line = file1.read()# Use this to read file content as a stream:
words = line.split()
for r in words:
    if not r in stop_words:
        appendFile = open('filteredtext.txt','a')
        appendFile.write(" "+r)
        appendFile.close()
```