distance

Une autre méthode pour comparer des chaînes de caractères de manière approximative est l'utilisation de l'algorithme de Jaccard. Cet algorithme mesure la similarité entre deux ensembles en calculant le rapport de la taille de leur intersection sur la taille de leur union.

Voici une version modifiée du code utilisant l'algorithme de Jaccard pour comparer les chaînes de caractères de manière plus rapide :

Sub ComparerTousLesElements()
Dim ws As Worksheet
Set ws = ThisWorkbook.Sheets("NomDeVotreFeuille") ' Remplacez par le nom de votre feuille

```
Dim derniereligne1 As Long
derniereligne1 = ws.Cells(ws.Rows.Count, "A").End(xlUp).Row  ' Supposons que
vos données sont dans la colonne A

Dim derniereligne2 As Long
derniereligne2 = ws.Cells(ws.Rows.Count, "B").End(xlUp).Row  ' Supposons que
vos données sont dans la colonne B

Dim colonne1 As Range
Set colonne1 = ws.Range("A1:A" & derniereligne1)

Dim colonne2 As Range
Set colonne2 = ws.Range("B1:B" & derniereligne2)

Dim seuil As Double
seuil = 0.8  ' Définissez votre seuil de correspondance ici

' Stocker les valeurs de la colonne 2 dans un tableau
Dim valeursColonne2() As Variant
valeursColonne2 = colonne2.Value

' Comparer chaque élément de la colonne 1 avec les valeurs stockées de la
colonne 2 en utilisant l'algorithme de Jaccard
Dim i As Long, j As Long
For i = 1 To derniereligne1
    For j = 1 To derniereligne2
        If JaccardSimilarity(colonne1.Cells(i, 1).Value, valeursColonne2(j,
1)) >= seuil Then
            ' Faites quelque chose lorsque les chaînes correspondent
```

```
approximativement, par exemple :
            ' ws.Cells(i, 3).Value = "Correspondance avec " &
valeursColonne2(j, 1)
            Exit For ' Sortez de la boucle interne si une correspondance est
trouvée
        End If
    Next j
Next i
```

End Sub

Function JaccardSimilarity(s1 As String, s2 As String) As Double
Dim set1 As Object, set2 As Object, intersectionSet As Object
Set set1 = CreateObject("Scripting.Dictionary")
Set set2 = CreateObject("Scripting.Dictionary")
Set intersectionSet = CreateObject("Scripting.Dictionary")

```
' Créer des ensembles (ensembles de caractères) pour les deux chaînes
For i = 1 To Len(s1)
    set1(Mid(s1, i, 1)) = 1
Next i

For i = 1 To Len(s2)
    set2(Mid(s2, i, 1)) = 1
Next i

' Trouver l'intersection des ensembles
For Each key In set1.Keys
    If set2.Exists(key) Then
        intersectionSet(key) = 1
    End If
Next key

' Calculer la similarité de Jaccard
JaccardSimilarity = intersectionSet.Count / (set1.Count + set2.Count -
intersectionSet.Count)
```

End Function

JACquard vs LEVES

La performance entre l'algorithme de Jaccard et l'algorithme de Levenshtein dépend du contexte spécifique, du jeu de données, et des exigences de votre application. Voici quelques considérations générales :

Complexité temporelle :

Levenshtein : La complexité temporelle de l'algorithme de Levenshtein est O(m * n), où m et n sont les longueurs des deux chaînes à comparer. Il peut être relativement coûteux pour de grandes chaînes.
Jaccard : La complexité temporelle dépend de la taille des ensembles impliqués. Calculer l'intersection et l'union des ensembles peut être plus rapide pour des chaînes plus courtes, mais cela peut aussi dépendre de la nature des caractères dans les chaînes.
Nature des données :

Levenshtein : Convient bien pour mesurer la distance d'édition entre deux chaînes, ce qui est utile pour des tâches comme la correction orthographique.
Jaccard : Convient lorsque la similarité entre ensembles est plus pertinente, par exemple, pour des tâches où l'ordre des caractères importe moins, comme la comparaison de mots-clés ou de tags.
Seuil de correspondance :

Levenshtein : Vous pouvez définir un seuil de correspondance pour déterminer si deux chaînes sont suffisamment similaires.
Jaccard : Vous pouvez également définir un seuil de similarité pour décider si deux ensembles sont suffisamment similaires.
Taille du jeu de données :

Levenshtein : Peut être plus coûteux pour de grandes quantités de données en raison de sa complexité temporelle.
Jaccard : Peut être plus efficace pour des jeux de données plus larges, mais cela dépend de la distribution des données.
En résumé, le choix entre l'algorithme de Jaccard et l'algorithme de Levenshtein dépend de la nature spécifique de votre tâche et des caractéristiques de vos données. Il est recommandé de tester ces algorithmes avec des données réelles et de mesurer leurs performances dans votre contexte spécifique pour prendre une décision éclairée.

Ces mesures de distance et de similitude entre chaînes de caractères sont toutes des méthodes couramment utilisées dans le domaine du traitement automatique du langage naturel et de la recherche d'informations pour comparer et mesurer la similarité entre deux chaînes. Chacune de ces méthodes a ses propres caractéristiques et applications spécifiques. Voici un bref aperçu de chacune :

Distance de Damerau-Levenshtein :

Mesure la distance d'édition entre deux chaînes en tenant compte des opérations d'insertion, de suppression, de substitution et de transposition.
Utile pour la correction orthographique et la comparaison de chaînes avec des erreurs typographiques.
Similitude de Jaro :

Mesure la similitude entre deux chaînes en se concentrant sur la fréquence des caractères communs et la proximité des positions de ces caractères.
Souvent utilisée pour la comparaison de noms ou d'entités.
Similitude de Jaro-Winkler :

Une extension de la similitude de Jaro qui donne un poids supplémentaire aux préfixes communs, pénalisant moins les petites différences au début des chaînes.
Convient pour les tâches où les préfixes communs sont significatifs.
Similitude de Smith-Waterman :

Utilisée pour trouver la meilleure correspondance locale entre deux chaînes.
Principalement employée dans la recherche de séquences similaires dans des contextes comme la bio-informatique.
Similitude de Sørensen-Dice :

Mesure la similitude entre deux ensembles en évaluant le rapport de la taille de leur intersection sur la taille de leur union.
Souvent utilisée pour la comparaison de textes courts et la recherche d'entités similaires.
Similitude avec Tversky :

Une mesure similaire à Sørensen-Dice mais avec des coefficients alpha et bêta qui permettent de pondérer différemment les faux positifs et les faux négatifs.
Utilisée dans la comparaison de documents et de séquences biologiques.
Similitude de chevauchement :

Mesure la similitude entre deux ensembles en évaluant le rapport de la taille de leur intersection sur la taille de l'ensemble plus petit.
Convient pour des tâches où la proportion d'éléments en commun est importante.
Similitude cosinus :

Utilise la similarité cosinus pour mesurer la similitude entre deux vecteurs représentant des documents ou des chaînes de caractères.
Couramment utilisée dans la recherche d'informations et le regroupement de documents.
Similitude N-gramme :

Mesure la similitude entre deux chaînes en considérant les sous-séquences de longueur n.
Utile dans la comparaison de chaînes avec des motifs similaires.
Similitude Ratcliff-Obershelp :

Mesure la similitude entre deux chaînes en utilisant une approche récursive basée sur la recherche des sous-chaînes communes.
Convient pour la comparaison de chaînes avec des séquences partiellement correspondantes.
Chacune de ces méthodes a ses avantages et ses limitations, et le choix dépend du contexte

spécifique de votre tâche et des caractéristiques de vos données. Il est souvent recommandé de tester plusieurs méthodes pour déterminer celle qui convient le mieux à votre cas d'utilisation particulier.

Le choix entre les différentes métriques de similarité et de dissimilarité dépend du contexte spécifique de votre tâche et des caractéristiques de vos données. La rapidité de ces métriques peut varier en fonction de plusieurs facteurs, notamment la longueur des chaînes à comparer, la nature des caractères, et la complexité algorithmique inhérente à chaque métrique.

Voici une brève évaluation des métriques que vous avez mentionnées en termes de rapidité générale :

Distance Damerau-Levenshtein : Elle est similaire à l'algorithme de Levenshtein, mais inclut également les transpositions de caractères. La complexité temporelle est généralement similaire à celle de Levenshtein.

Similitude avec Jaro : L'algorithme de Jaro mesure la similarité entre deux chaînes et peut être relativement rapide. Il est particulièrement adapté pour les chaînes de caractères de longueur similaire.

Similitude Jaro-Winkler : Cette mesure ajoute une pondération pour les préfixes communs et peut être plus rapide que le Jaro seul dans certains cas.

Similitude Smith-Waterman : C'est un algorithme de comparaison de séquences qui peut être plus coûteux en termes de calculs, surtout pour des séquences longues.

Similitude Sørensen-Dice : Elle mesure la similarité entre deux ensembles, et sa rapidité dépend de la taille des ensembles à comparer.

Similitude avec Tversky : C'est une métrique de similarité qui prend en compte les faux positifs et les faux négatifs. Sa rapidité dépend de la complexité des opérations impliquées.

Similitude de chevauchement : Mesure la taille de l'intersection par rapport à la plus petite des deux chaînes. Son efficacité dépend de la taille des ensembles.

Similitude cosinus : Mesure la similarité entre deux vecteurs. La rapidité dépend de la dimension des vecteurs.

Similitude N-gramme : Mesure la similitude entre deux chaînes basées sur les sous-chaînes de longueur fixe (N-grammes). La rapidité dépend de la longueur des chaînes.

Similitude Ratcliff-Obershelp : Un algorithme de comparaison de séquences basé sur la recherche de sous-séquences communes. Sa rapidité dépend de la longueur des séquences.

Il est important de noter que la rapidité n'est qu'un des nombreux critères à considérer lors du choix d'une métrique. La précision, la sensibilité aux types d'erreurs que vous souhaitez traiter, la tolérance aux variations, et d'autres facteurs peuvent également être cruciaux en fonction de votre application

spécifique. Il est recommandé de tester différentes métriques avec vos données pour déterminer celle qui répond le mieux à vos besoins particuliers.

Bien sûr, voici une implémentation simple en VBA de la distance de Damerau-Levenshtein. Cette fonction renvoie le nombre minimum d'opérations (insertions, suppressions, substitutions et transpositions) nécessaires pour transformer une chaîne source en une chaîne cible :

```
Function DistanceDamerauLevenshtein(s1 As String, s2 As String) As Integer
Dim len1 As Integer, len2 As Integer
Dim matrix() As Integer
Dim i As Integer, j As Integer, cost As Integer

len1 = Len(s1)
len2 = Len(s2)

ReDim matrix(0 To len1, 0 To len2)

For i = 0 To len1
    matrix(i, 0) = i
Next

For j = 0 To len2
    matrix(0, j) = j
Next

For i = 1 To len1
    For j = 1 To len2
        If Mid(s1, i, 1) = Mid(s2, j, 1) Then
            cost = 0
        Else
            cost = 1
        End If

        matrix(i, j) = WorksheetFunction.Min3( _
            matrix(i - 1, j) + 1, _
            matrix(i, j - 1) + 1, _
            matrix(i - 1, j - 1) + cost _
        )

        If i > 1 And j > 1 And Mid(s1, i, 1) = Mid(s2, j - 1, 1) And Mid(s1, i - 1, 1) = Mid(s2, j, 1) Then
            matrix(i, j) = WorksheetFunction.Min(matrix(i, j), matrix(i - 2, j - 2) + cost)
```

```
        End If
    Next
Next

DistanceDamerauLevenshtein = matrix(len1, len2)
```

End Function

Sub TestDistanceDamerauLevenshtein()
Dim str1 As String
Dim str2 As String

```
str1 = "kitten"
str2 = "sitting"

MsgBox "Distance Damerau-Levenshtein entre '" & str1 & "' et '" & str2 & "'
: " & DistanceDamerauLevenshtein(str1, str2)
```

End Sub

Voici une implémentation simple en VBA de la métrique de similarité Jaro-Winkler :

Function SimilarityJaroWinkler(s1 As String, s2 As String) As Double
Dim matchWindow As Integer
Dim commonChars As Integer
Dim transpositions As Integer
Dim prefixLength As Integer
Dim maxPrefixLength As Integer
Dim jaroSimilarity As Double

```
' Longueur de la fenêtre de correspondance
matchWindow = Int(Len(s1) / 2) - 1

' Recherche des caractères communs
commonChars = CountCommonChars(s1, s2, matchWindow)

' Calcul du nombre de transpositions
transpositions = CountTranspositions(s1, s2, matchWindow)

' Calcul de la longueur du préfixe commun
prefixLength = CountCommonPrefix(s1, s2)

' Détermination de la longueur maximale du préfixe (jusqu'à 4 caractères)
```

```
maxPrefixLength = IIf(prefixLength > 4, 4, prefixLength)

' Calcul de la similarité Jaro
jaroSimilarity = JaroSimilarity(s1, s2, commonChars, transpositions,
prefixLength)

' Calcul de la métrique Jaro-Winkler
SimilarityJaroWinkler = jaroSimilarity + 0.1 * maxPrefixLength * (1 -
jaroSimilarity)
```

End Function

Function CountCommonChars(s1 As String, s2 As String, matchWindow As Integer) As Integer
Dim commonChars As Integer
Dim s2Matched As Boolean

```
commonChars = 0
s2Matched = False

For i = 1 To Len(s1)
    For j = WorksheetFunction.Max(1, i - matchWindow) To
WorksheetFunction.Min(Len(s2), i + matchWindow)
        If Mid(s1, i, 1) = Mid(s2, j, 1) And Not s2Matched Then
            commonChars = commonChars + 1
            s2Matched = True
            Exit For
        End If
    Next j
    s2Matched = False
Next i

CountCommonChars = commonChars
```

End Function

Function CountTranspositions(s1 As String, s2 As String, matchWindow As Integer) As Integer
Dim transpositions As Integer

```
transpositions = 0

For i = 1 To Len(s1)
    If Mid(s1, i, 1) <> Mid(s2, i, 1) Then
        For j = WorksheetFunction.Max(1, i - matchWindow) To
WorksheetFunction.Min(Len(s2), i + matchWindow)
```

```
            If Mid(s1, i, 1) = Mid(s2, j, 1) Then
                transpositions = transpositions + 1
                Exit For
            End If
        Next j
    End If
Next i

CountTranspositions = transpositions / 2
```

End Function

Function CountCommonPrefix(s1 As String, s2 As String) As Integer
Dim commonPrefix As Integer

```
commonPrefix = 0

For i = 1 To WorksheetFunction.Min(Len(s1), Len(s2))
    If Mid(s1, i, 1) = Mid(s2, i, 1) Then
        commonPrefix = commonPrefix + 1
    Else
        Exit For
    End If
Next i

CountCommonPrefix = commonPrefix
```

End Function

Function JaroSimilarity(s1 As String, s2 As String, commonChars As Integer, transpositions As Integer, prefixLength As Integer) As Double
Dim m As Double
Dim t As Double
Dim l As Double

```
m = CDbl(commonChars)
t = CDbl(transpositions)
l = CDbl(prefixLength)

If m = 0 Then
    JaroSimilarity = 0
Else
    JaroSimilarity = (m / Len(s1) + m / Len(s2) + (m - t) / m) / 3
End If
```

End Function

Sub TestSimilarityJaroWinkler()
Dim str1 As String
Dim str2 As String

```
str1 = "DIXON"
str2 = "DICKSONX"

MsgBox "Similarité Jaro-Winkler entre '" & str1 & "' et '" & str2 & "' : " &
SimilarityJaroWinkler(str1, str2)
```

End Sub

Adaptez ce code en fonction de vos besoins spécifiques. Notez que la similarité Jaro-Winkler a une plage de valeurs de 0 à 1, où 1 indique une correspondance parfaite. La pondération du préfixe commun (0.1 * maxPrefixLength * (1 - jaroSimilarity)) est utilisée pour donner plus de poids aux préfixes communs. Vous pouvez ajuster ces paramètres selon vos préférences.

Le choix entre la distance de Damerau-Levenshtein et la métrique Jaro-Winkler dépend de la nature spécifique de votre tâche et des caractéristiques de vos données. Voici quelques considérations pour vous aider à prendre une décision :

Distance de Damerau-Levenshtein :

Avantages :

Mesure le nombre minimum d'opérations nécessaires pour transformer une chaîne en une autre (insertions, suppressions, substitutions, transpositions).
Utile pour des tâches où la distance d'édition exacte est importante, par exemple dans la correction orthographique.
Inconvénients :

Peut être coûteux en termes de calculs, surtout pour des chaînes de caractères longues.
Peut ne pas bien capturer la similarité sémantique entre les chaînes.
Jaro-Winkler :

Avantages :

Conçu pour mesurer la similarité entre deux chaînes en tenant compte des transpositions.
Donne un poids supplémentaire aux préfixes communs, ce qui peut être utile dans certains cas.
Inconvénients :

Peut être sensible à la longueur des chaînes et peut ne pas bien fonctionner pour des chaînes très courtes.
Ne mesure pas le nombre d'opérations éditées, ce qui peut être un inconvénient dans certaines applications.
En termes de rapidité, la Jaro-Winkler est généralement plus rapide que la Damerau-Levenshtein, surtout pour des chaînes de caractères de longueur modérée. Cependant, cela dépend également des implémentations spécifiques et de la nature des caractères dans vos données.

Voici une implémentation simple en VBA de la métrique Jaro-Winkler :

```
Function JaroWinklerSimilarity(s1 As String, s2 As String) As Double
Dim jaroSimilarity As Double
Dim commonPrefix As Integer
Dim prefixScale As Double

jaroSimilarity = JaroSimilarity(s1, s2)

' Calcul du préfixe commun et application de la pondération Jaro-Winkler
commonPrefix = CommonPrefixLength(s1, s2)
prefixScale = 0.1  ' Ajustez la pondération selon vos besoins

JaroWinklerSimilarity = jaroSimilarity + commonPrefix * prefixScale * (1 -
jaroSimilarity)

End Function

Function JaroSimilarity(s1 As String, s2 As String) As Double
Dim m As Integer
Dim t As Integer
Dim l As Integer
Dim p As Double

l = WorksheetFunction.Max(Len(s1), Len(s2)) \ 2 - 1

m = MatchingCharacters(s1, s2, l)
t = Transpositions(s1, s2)

If m = 0 Then
    JaroSimilarity = 0
Else
    p = 0.1  ' Ajustez la pondération selon vos besoins
    JaroSimilarity = (m / Len(s1) + m / Len(s2) + (m - t) / m) / 3 * (1 - p)
+ p
End If
```

End Function

Function MatchingCharacters(s1 As String, s2 As String, l As Integer) As Integer
Dim common As String
Dim match As Boolean
Dim i As Integer, j As Integer

```
common = ""
match = False

For i = 1 To Len(s1)
    For j = WorksheetFunction.Max(1, i - l) To
WorksheetFunction.Min(Len(s2), i + l)
        If Mid(s1, i, 1) = Mid(s2, j, 1) Then
            match = True
            common = common & Mid(s1, i, 1)
            Exit For
        End If
    Next j
Next i

MatchingCharacters = Len(common)
```

End Function

Function CommonPrefixLength(s1 As String, s2 As String) As Integer
Dim minLength As Integer
minLength = WorksheetFunction.Min(Len(s1), Len(s2))

```
Dim i As Integer
For i = 1 To minLength
    If Mid(s1, i, 1) <> Mid(s2, i, 1) Then
        Exit For
    End If
Next i

CommonPrefixLength = i - 1
```

End Function

Function Transpositions(s1 As String, s2 As String) As Integer
Dim transpositions As Integer
Dim common As String
Dim i As Integer

```
transpositions = 0
common = ""

For i = 1 To Len(s1)
    If Mid(s1, i, 1) = Mid(s2, i, 1) Then
        common = common & Mid(s1, i, 1)
    Else
        transpositions = transpositions + 1
    End If
Next i

' Compter le nombre de transpositions
For i = 1 To Len(common) Step 2
    If Mid(common, i, 1) <> Mid(common, i + 1, 1) Then
        transpositions = transpositions + 1
    End If
Next i

Transpositions = transpositions \ 2
```

End Function

medium

https://yassineelkhal.medium.com/the-complete-guide-to-string-similarity-algorithms-1290ad07c6b7

Conclusion
En conclusion, le monde des algorithmes de similarité de chaînes ouvre un domaine fascinant de possibilités pour diverses applications dans le traitement du langage naturel, l'analyse de données et au-delà. Tout au long de cet article, nous avons exploré de manière approfondie trois catégories fondamentales d'algorithmes : basés sur l'édition, basés sur des jetons et basés sur des séquences. Les points importants à retenir sont :

Algorithmes basés sur l'édition :

Les distances de Damerau-Levenshtein et Jaro Winkler quantifient la similarité au niveau des caractères grâce à des opérations d'édition.
Idéal pour les applications mettant l'accent sur les modifications et les corrections d'un seul caractère.
Utile pour les correcteurs orthographiques, l'OCR, la précision des caractères et la correction automatique du texte.
Algorithmes basés sur des jetons :

Les similitudes entre Jaccard et Sørensen-Dice se concentrent sur les ensembles de jetons, sans tenir compte de la séquence.

Efficace pour le regroupement de documents, la détection du plagiat et les systèmes de recommandation.
Mettez l’accent sur la présence plutôt que sur l’ordre, ce qui facilite la catégorisation.
Peut être utilisé pour la catégorisation du contenu et la comparaison de documents.
Algorithmes basés sur des séquences :

Les méthodes LCS (Longest Common Subsequence) et Ratcliff/Obershelp préservent l'ordre séquentiel.
Crucial pour l’alignement des séquences d’ADN, la détection des quasi-doublons de texte et le contrôle des versions.
Offrez un aperçu des relations structurelles au sein des chaînes.
En parcourant les subtilités de ces approches algorithmiques, il est devenu évident qu’aucune méthode ne convient à tous les scénarios. Chaque catégorie possède ses atouts, répondant à des besoins et à des contextes spécifiques. Un bon data scientist possédant la connaissance de ces diverses techniques peut exploiter leur puissance combinée pour relever un large éventail de défis liés à l’analyse de similarité de chaînes.