

TEXT SIMILARITY

- Text Similarity is the process of comparing a piece of text with another and finding the similarity between them.
- Text similarity means user's query text is matched with the document text and on the basis of this matching user retrieves the most relevant documents.
- Its possible to measure the similarity between sentences, words, paragraphs and documents to categorize them in an efficient way.
- It is defined or classified in two ways; these are lexical similarity and semantic similarity.
- Lexical similarity provide the similarity on the basis of character or statement matching.

sentence1 = "I love machine learning", sentence2 = "Machine learning is amazing".

• **Common words** = {"machine", "learning"}

• **Total unique words** = {"I", "love", "machine", "learning", "is", "amazing"}

• **Similarity Score** = Number of Common Words / Average Length of Sentences (in words)
= $2/4 = 0.50$ (50%)

- Semantic similarity provide the similarity on the basis of meaning, for e.g. "Support Vector Machine" and "SVM" both are semantic similar to each other.



TEXT SIMILARITY

- There are several applications or areas where we use the text similarity; these areas are Information retrieval, clustering, text categorization, topic detection, question answer session, machine translation, text summarization etc.

1. LEXICAL SIMILARITY

- This involves observing the contents of the text documents with regard to syntax, structure, content and measuring their similarity based on these parameters.
- Lexical similarity is a measure of the degree to which word set of two given string are similar.
- A Lexical similarity of 1(means 100%) would mean a total overlap between words, whereas Lexical similarity of 0 means there are no common word in given string.



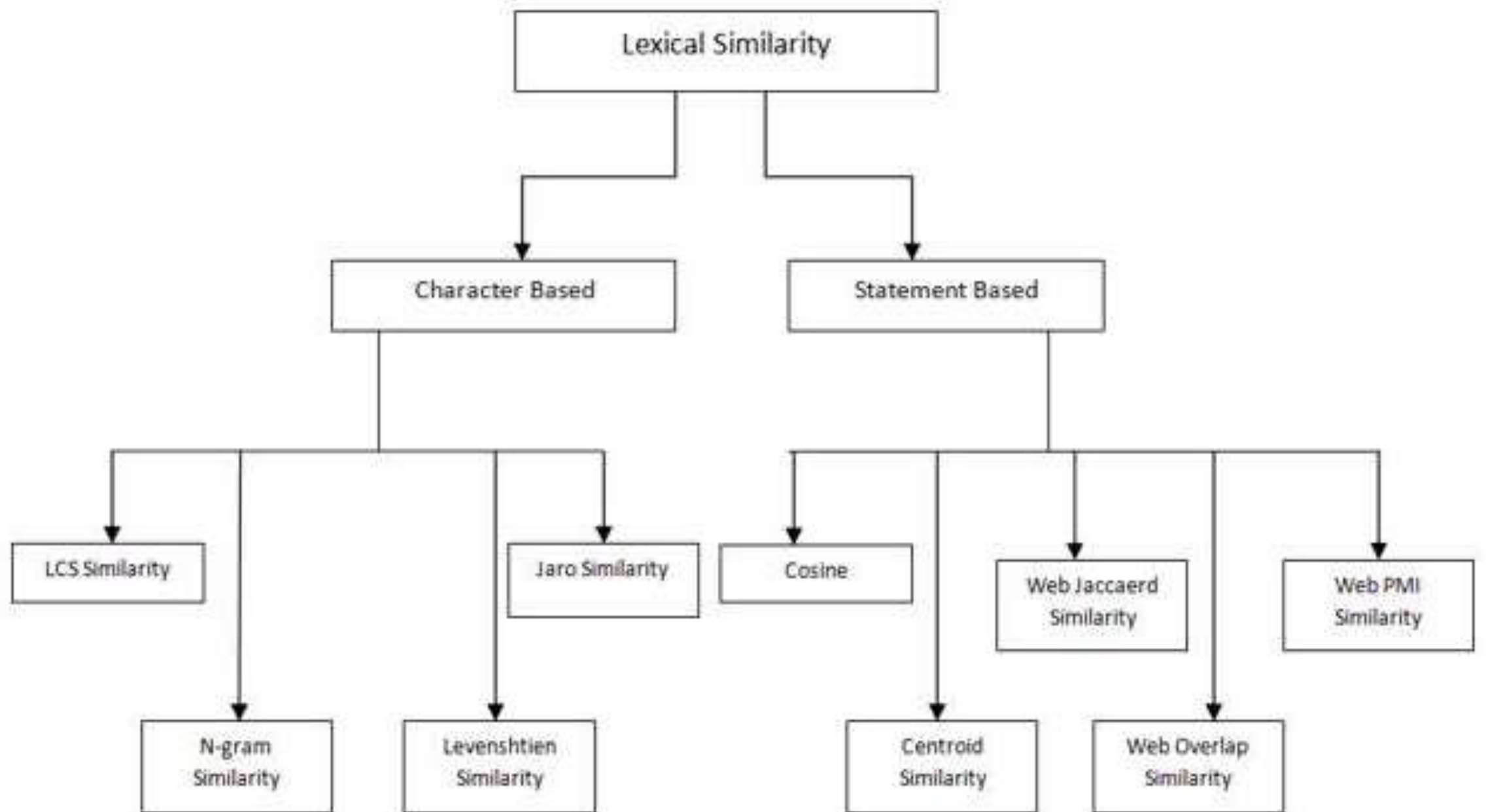


Figure 1: Categorization of Lexical Similarity

LEXICAL SIMILARITY

i. Longest common subsequence (LCS) Similarity

- String matching is a commonly used technique to measure the similarity between two strings (i, j). LCS is the longest subsequence that appears in the same order in both strings but not necessarily continuously.
- LCS measure the longest total length of all the matched substring between two string where these sub-string appear in the same order as they appear in the other string.
- LCS similarity of Given Two string (i, j) will be

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS(i - 1, j - 1) & \text{if } x[i] == y[j] \\ \max \begin{cases} LCS(i, j - 1) \\ LCS(i - 1, j) \end{cases} & \text{if } x[i] \neq y[j] \end{cases}$$

- String 1: "ABCDEF"

- String 2: "AEBDF"

- 'A' (✓ Match) → Both strings start with 'A'.
- 'B' (✓ Match) → 'B' appears after 'A' in both.
- 'C' (✗ No match) → 'C' is in **String 1** but not in **String 2**, so we skip it.
- 'D' (✓ Match) → 'D' appears after 'B' in both.
- 'E' (✗ No match) → 'E' in **String 2** appears before 'B', so we ignore it
- 'F' (✓ Match) → 'F' appears at the end of both strings.



- Longest Common Subsequence (LCS) = "ABDF"

- Length = 4

LEXICAL SIMILARITY

ii. N-gram similarity

- N-Gram similarity measures how similar two texts are by **breaking them into N-sized chunks (n-grams)** and comparing the overlap.
- In this, we compute the similarity on the basis of distance between each character in two strings.
- distance is computed by dividing the number of similar grams by maximal number of n-grams.

$$P(w) = P(w_1)P(w_2/w_1)P(w_3/w_2, w_1) \dots \dots \dots$$

Where w, w1, w2, w3 are different words. The N-gram similarity technique is used to design kernels that allow ML algorithms such as support vector machine to learn from string data.

Sentence 1: "hello world"

Sentence 2: "hello there"

Step 2: Create Bigrams (2-Grams)

Sentence 1 ("hello world") → ['he', 'el', 'll', 'lo', 'o ', ' w', 'wo', 'or', 'rl', 'ld']

Sentence 2 ("hello there") → ['he', 'el', 'll', 'lo', 'o ', ' t', 'th', 'he', 'er', 're']

Step 3: Find Common Bigrams

Common bigrams: ['he', 'el', 'll', 'lo', 'o ']

Total unique bigrams: 15

Step 4: Calculate Similarity

Similarity=Common Bigrams/Total Unique Bigrams=5/15=0.33(33%)



Statement based similarity

i. Cosine similarity

- Cosine similarity is widely used approach to find the similarity between two texts were, each text is represented in the form of vector.
- Measures how **similar two texts or vectors** are based on their **angle** rather than their magnitude.
- Each word in text defines a dimension in Euclidean space and the frequency of each word corresponds to the value in the dimension.
- The Cosine similarity between two text (t_1, t_2)

$$SIM(t_1, t_2) = \frac{\sum_{i=1}^n t_{1_i} t_{2_i}}{\sqrt{\sum t_{1_i}^2} \times \sqrt{\sum t_{2_i}^2}}$$



EXAMPLE OF COSINE SIMILARITY

Step 1: Sentence 1: "I love machine learning"

Sentence 2: "Machine learning is amazing"

Step 2: Convert Sentences to Word Vectors

Sentence 1 \rightarrow {I:1, love:1, machine:1, learning:1, is:0, amazing:0}

Sentence 2 \rightarrow {I:0, love:0, machine:1, learning:1, is:1, amazing:1}

Word	I	love	machine	learning	is	amazing
Sentence 1	1	1	1	1	0	0
Sentence 2	0	0	1	1	1	1

Step 3: Compute Cosine Similarity

$$\begin{aligned}\text{Cosine Similarity} &= \frac{A \cdot B}{||A|| \times ||B||} \\&= \frac{(1 \times 0 + 1 \times 0 + 1 \times 1 + 1 \times 1 + 0 \times 1 + 0 \times 1)}{\sqrt{(1^2 + 1^2 + 1^2 + 1^2 + 0^2 + 0^2)} \times \sqrt{(0^2 + 0^2 + 1^2 + 1^2 + 1^2 + 1^2)}} \\&= \frac{(0 + 0 + 1 + 1 + 0 + 0)}{\sqrt{(1 + 1 + 1 + 1 + 0 + 0)} \times \sqrt{(0 + 0 + 1 + 1 + 1 + 1)}} \\&= \frac{2}{\sqrt{4} \times \sqrt{4}} = \frac{2}{4} = 0.5\end{aligned}$$

Cosine Similarity = 0.5 (50% similarity)



SEMANTIC SIMILARITY

- Semantic similarity refers to the similarity of two pieces of text when their contextual meaning is considered. It judges the order of occurrences of the words in the text.

Ex: "happy" and "joyful" are semantically similar

Types of Semantic similarity

1. **Knowledge-Based Similarity** — Determining semantic similarity between the concept of corpus.
2. **Statistical-Based Similarity** — Determining semantic similarity based on learning features' vectors from the corpus.
3. **String-Based Similarity** — Combines the above two approaches to find the similarity between non-zero vectors.

 String-based similarity measures how similar two strings are based on their characters and substrings. Methods like spell checking.

ANALYZING TERM SIMILARITY

- term similarity refers to the similarity between individual word or tokens.
- several applications and use-cases like autocompleters, spell check, and correctors use some of these techniques to correct misspelled terms.
- In this, couple of words will be taken and the similarity between them will be measured using different word representations as well as distance metrics.
- The word representations we will be using are as follows:
 - ❑ Character vectorization
 - ❑ Bag of Characters vectorization



For character vectorization, it is an extremely simple process of just mapping each character of the term to a corresponding unique number. We can do that using the function depicted in the following snippet:

```
import numpy as np

def vectorize_terms(terms):
    terms = [term.lower() for term in terms]
    terms = [np.array(list(term)) for term in terms]
    terms = [np.array([ord(char) for char in term])
              for term in terms]

    return terms
```

The function takes input a list of words or terms and returns the corresponding character vectors for the words.



Bag of Characters vectorization is very similar to the Bag of Words model except here we compute the frequency of each character in the word. Sequence or word orders are not taken into account. The following function helps in computing this:

```
from scipy.stats import itemfreq
```

```
def boc_term_vectors(word_list):
```

```
word_list = [word.lower() for word in word_list]
```

```
unique_chars = np.unique(np.hstack([list(word) for word in word_list]))
```

```
word_list_term_counts = [{char: count for char, count in itemfreq(list(word))}]
```



```
for word in word_list]
boc_vectors = [np.array([int(word_term_counts.get(char, 0))
for char in unique_chars])
for word_term_counts in word_list_term_counts]
return list(unique_chars), boc_vectors
```

In that function, we take in a list of words or terms and then extract the unique characters from all the words. This becomes our feature list, just like we do in Bag of Words, where instead of characters, unique words are our features. Once we have this list of unique_chars, we get the count for each of the characters in each word and build our Bag of Characters vectors.



USING A TOTAL OF FOUR EXAMPLE TERMS AND COMPUTING THE SIMILARITY AMONG THEM

```
root = 'Believe'
```

```
term1 = 'beleive'
```

```
term2 = 'bargain'
```

```
term3 = 'Elephant'
```

```
terms = [root, term1, term2, term3]
```

```
# Character vectorization
```

```
vec_root, vec_term1, vec_term2, vec_term3 = vectorize_terms(terms)
```

```
# show vector representations
```

```
In [103]: print '''
```

```
...: root: {}
```

```
...: term1: {}
```

```
...: term2: {}
```

```
...: term3: {}
```

```
...: ".format(vec_root, vec_term1, vec_term2, vec_term3)
```

```
root: [ 98 101 108 105 101 118 101]
```

```
term1: [ 98 101 108 101 105 118 101]
```

```
term2: [ 98 97 114 103 97 105 110]
```

```
term3: [101 108 101 112 104 97 110 116]
```

```
# Bag of characters vectorization
```

```
features, (boc_root, boc_term1, boc_term2, boc_term3) = boc_term_  
vectors(terms)
```

```
# show features and vector representations
```



```
In [105]: print 'Features:', features
```

```
...: print "
```

```
...: root: {}
```

```
...: term1: {}
```

```
...: term2: {}
```

```
...: term3: {}
```

```
...: """.format(boc_root, boc_term1, boc_term2, boc_term3)
```

```
Features: ['a', 'b', 'e', 'g', 'h', 'i', 'l', 'n', 'p', 'r', 't', 'v']
```

```
root: [0 1 3 0 0 1 1 0 0 0 0 1]
```

```
term1: [0 1 3 0 0 1 1 0 0 0 0 1]
```

```
term2: [2 1 0 1 0 1 0 1 0 1 0 0]
```

```
term3: [1 0 2 0 1 0 1 1 1 0 1 0]
```

 we can easily transform text terms into numeric vector representations.

SIMILARITY USING DISTANCE METRICS

- using several distance metrics to compute similarity between the root word and the other three words mentioned in the preceding snippet.
- There are a lot of distance metrics out there that you can use to compute and measure similarities:
 - ☐ Hamming distance
 - ☐ Manhattan distance
 - ☐ Euclidean distance
 - ☐ Levenshtein edit distance
 - ☐ Cosine distance and similarity



First, some necessary variables needs to be set storing the root term, the other terms with which its similarity will be measures, and their various vector representations using the following snippet:

```
root_term = root
```

```
root_vector = vec_root
```

```
root_boc_vector = boc_root
```

```
terms = [term1, term2, term3]
```

```
vector_terms = [vec_term1, vec_term2, vec_term3]
```

```
boc_vector_terms = [boc_term1, boc_term2, boc_term3]
```

Now we can compute the similarity metrics.



HAMMING DISTANCE

- The Hamming distance is a very popular distance metric used frequently in information theory and communication systems.
- It is distance measured between two strings under the assumption that they are of equal length.
- Formally, it is defined as the number of positions that have different characters or symbols between two strings of equal length.
- Considering two terms u and v of length n , we can mathematically denote Hamming distance as

$$hd(u, v) = \sum_{i=1}^n (u_i \neq v_i)$$

we can also normalize it if you want by dividing the number of mismatches by the total length of the terms to give the normalized hamming distance, which is represented as

$$norm_hd(u, v) = \frac{\sum_{i=1}^n (u_i \neq v_i)}{n}$$

n denotes the length of the terms.



EXAMPLE

Let's say you want to calculate the Hamming Distance between:

- str1 = "karolin"
- str2 = "kathrin"

Step-by-Step Comparison:

Position	str1	str2	Same?
1	k	k	✓
2	a	a	✓
3	r	t	✗
4	o	h	✗
5	l	r	✗
6	i	i	✓
7	n	n	✓

•Differences at positions 3, 4, and 5.

•**Hamming Distance = 3**



FUNCTION TO COMPUTE HAMMING DISTANCE BETWEEN TWO TERMS AND NORMALIZED HAMMING DISTANCE

```
def hamming_distance(u, v, norm=False):  
    if u.shape != v.shape:  
        raise ValueError('The vectors must have equal lengths.')  
    return (u != v).sum() if not norm else (u != v).mean()
```

To measure the Hamming distance between our root term and the other terms

```
# compute Hamming distance  
for term, vector_term in zip(terms, vector_terms):  
    ...: print 'Hamming distance between root: {} and term: {} is {}'.  
format(root_term, ...: term, hamming_distance(root_vector, vector_term,  
norm=False))
```

Hamming distance between root: Believe and term: beleive is 2

Hamming distance between root: Believe and term: bargain is 6

Traceback (most recent call last):

File "<ipython-input-115-3391bd2c4b7e>", line 4, in <module>

hamming_distance(root_vector, vector_term, norm=False))

ValueError: The vectors must have equal lengths.

compute normalized Hamming distance

for term, vector_term in zip(terms, vector_terms):

...: print 'Normalized Hamming distance between root: {} and term: {} is ...:

{ {}'.format(root_term, term, ...: round(hamming_distance(root_vector, vector_term, norm=True), 2))



Normalized Hamming distance between root: Believe and term: beleive is 0.29

Normalized Hamming distance between root: Believe and term: bargain is 0.86

Traceback (most recent call last):

File "<ipython-input-117-7dfc67d08c3f>", line 4, in <module>

round(hamming_distance(root_vector, vector_term, norm=True), 2))

ValueError: The vectors must have equal lengths






from the preceding output, it can be seen that terms '**Believe**' and '**beleive**' ignoring their case are most similar to each other with the Hamming distance of 2 or 0.29, compared to the term '**bargain**' giving scores of 6 or 0.86 (here, the smaller the score, the more similar are the terms).

The term '**Elephant**' throws an exception because the length of that term (term3) is 8 compared to length 7 of the root term '**Believe**', hence Hamming distance can't be computed because the base assumption of strings being of equal length is violated.



MANHATTAN DISTANCE

- The Manhattan distance metric is similar to the Hamming distance conceptually, where instead of counting the number of mismatches, we subtract the difference between each pair of characters at each position of the two strings.
- Formally, Manhattan distance is also known as city block distance, L1 norm, taxicab metric and is defined as the distance between two points in a grid based on strictly horizontal or vertical paths instead of the diagonal distance conventionally calculated by the Euclidean distance metric.
- Mathematically it can be denoted as


$$norm_md(u, v) = \frac{\|u - v\|_1}{n} = \frac{\sum_{i=1}^n |u_i - v_i|}{n}$$

where n is the length of each of the terms u and v .

EXAMPLE

- Sentence 1: "I love NLP"
- Sentence 2: "I enjoy learning NLP"

1. Vocabulary:

["I", "love", "NLP", "enjoy", "learning"]

2. Vectors (word count):

- Sentence 1 → [1, 1, 1, 0, 0]
- Sentence 2 → [1, 0, 1, 1, 1]

3. Manhattan Distance:

$$|1 - 1| + |1 - 0| + |1 - 1| + |0 - 1| + |0 - 1| = 0 + 1 + 0 + 1 + 1 = \boxed{3}$$



FUNCTION TO IMPLEMENT MANHATTAN DISTANCE AND NORMALIZED MANHATTAN DISTANCE

```
def manhattan_distance(u, v, norm=False):  
    if u.shape != v.shape:  
        raise ValueError('The vectors must have equal lengths.')  
    return abs(u - v).sum() if not norm else abs(u - v).mean()
```



The Manhattan distance between our root term and the other terms using the previous function can be computed using:

```
# compute Manhattan distance
```

```
for term, vector_term in zip(terms, vector_terms):
```

```
...: print 'Manhattan distance between root: {} and term: {} is
```

```
{ {}'.format(root_term,
```

```
...: term, manhattan_distance(root_vector,
```

```
vector_term, norm=False))
```

Manhattan distance between root: Believe and term: beleive is 8

Manhattan distance between root: Believe and term: bargain is 38

Traceback (most recent call last):

File "<ipython-input-120-b228f24ad6a2>", line 4, in <module>

manhattan_distance(root_vector, vector_term, norm=False))

ValueError: The vectors must have equal lengths.

```
# compute normalized Manhattan distance
```

```
In [122]: for term, vector_term in zip(terms, vector_terms):
```

```
...: print 'Normalized Manhattan distance between root: {} and  
term: {} is {}'.format(root_term,
```

```
...: term,
```

```
...: round(manhattan_distance(root_vector, vector_term,  
norm=True),2))
```

```
...:
```

```
...:
```

```
1. Normalized Manhattan distance between root: Believe and term: beleive is 1.14
```

```
2. Normalized Manhattan distance between root: Believe and term: bargain is 5.43
```

```
3. Traceback (most recent call last):
```

```
4. File "<ipython-input-122-d13a48d56a22>", line 4, in <module>
```

```
5. round(manhattan_distance(root_vector, vector_term, norm=True),2))
```

```
6. ValueError: The vectors must have equal lengths.
```



- the distance between 'Believe' and 'believe' ignoring their case is most similar to each other, with a score of 8 or 1.14, as compared to 'bargain', which gives a score of 38 or 5.43 (here the smaller the score, the more similar the words).
- The term 'Elephant' yields an error because it has a different length compared to the base term just as we noticed earlier when computing Hamming distances.



EUCLIDEAN DISTANCE

- Formally, the *Euclidean distance* is also known as the *Euclidean norm*, *L2 norm*, or *L2 distance*
- It is defined as the shortest straight-line distance between two points. Mathematically this can be denoted as

$$ed(u, v) = \|u - v\|_2 = \sqrt{\sum_{i=1}^n (u_i - v_i)^2}$$

where the two points u and v are vectorized text terms in our scenario, each having length n .



EXAMPLE

- Sentence 1: "I love NLP"
- Sentence 2: "I enjoy learning NLP"

1. Vocabulary:

["I", "love", "NLP", "enjoy", "learning"]

2. Vectors (word count):

- Sentence 1 → [1, 1, 1, 0, 0]
- Sentence 2 → [1, 0, 1, 1, 1]

▢ Euclidean Distance Formula:

$$\text{Distance} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$

🌈 Calculation:

$$\begin{aligned} & \sqrt{(1-1)^2 + (1-0)^2 + (1-1)^2 + (0-1)^2 + (0-1)^2} \\ &= \sqrt{0+1+0+1+1} = \sqrt{3} \approx \boxed{1.732} \end{aligned}$$



FUNCTION TO COMPUTE EUCLIDEAN DISTANCE BETWEEN TWO TERMS

```
def euclidean_distance(u, v):  
    if u.shape != v.shape:  
        raise ValueError('The vectors must have equal lengths.')  
    distance = np.sqrt(np.sum(np.square(u - v)))  
    return distance
```



To compare the Euclidean distance among our terms by using the preceding function:

```
# compute Euclidean distance
```

```
In [132]: for term, vector_term in zip(terms, vector_terms):
```

```
...: print 'Euclidean distance between root: {} and term: {} is
```

```
{ {}'.format(root_term,
```

```
...: term, round(euclidean_distance(root_  
vector, vector_term),2))
```

```
Euclidean distance between root: Believe and term: beleive is 5.66
```

```
Euclidean distance between root: Believe and term: bargain is 17.94
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-132-90a4dbe8ce60>", line 4, in <module>
```

```
round(euclidean_distance(root_vector, vector_term),2))
```

```
ValueError: The vectors must have equal lengths.
```



- the terms 'Believe' and 'believe' are the most similar with a score of 5.66 compared to 'bargain' giving us a score of 17.94, and 'Elephant' throws a ValueError because the base assumption that strings being compared should have equal lengths holds good for this distance metric also.
- So far, all the distance metrics we have used work on strings or terms of the same length and fail when they are not of equal length.
- So how do we deal with this problem?
- We will now look at a couple of distance metrics that work even with strings of unequal length to measure similarity.



LEVENSHTEIN EDIT DISTANCE

- The *Levenshtein edit distance*, often known as just Levenshtein distance, belongs to the family of edit distance–based metrics.
- It is used to measure the distance between two sequence of strings based on their differences—similar to the concept behind Hamming distance.
- The Levenshtein edit distance between two terms can be defined as the minimum number of edits needed in the form of additions, deletions, or substitutions to change or convert one term to the other.

The allowed operations are:

1.Insertion (e.g., changing "cat" to "cats" by inserting 's')

2.Deletion (e.g., changing "cats" to "cat" by deleting 's')

3.Substitution (e.g., changing "cat" to "cut" by substituting 'a' with 'u')

- These substitutions are character-based substitutions, where a single character can be edited in a single operation.



- Also, as mentioned before, the length of the two terms need not be equal here.

- Mathematically, we can represent the Levenshtein edit distance between two terms as $ld_{u,v}(|u|, |v|)$ such that u and v are our two terms where $|u|$ and $|v|$ are their lengths. This distance can be represented by the following formula

$$ld_{u,v}(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0 \\ \min \begin{cases} ld_{u,v}(i-1, j) + 1 \\ ld_{u,v}(i, j-1) + 1 \\ ld_{u,v}(i-1, j-1) + C_{u_i \neq v_j} \end{cases} & \text{otherwise} \end{cases}$$

where i and j are basically indices for the terms u and v . The third equation in the minimum above has a cost function denoted by $C_{u_i \neq v_j}$ that it has the following conditions

$$C_{u_i \neq v_j} = \begin{cases} 1 & \text{if } u_i \neq v_j \\ 0 & \text{if } u_i = v_j \end{cases}$$

and this denotes the indicator function, which depicts the cost associated with two characters being matched for the two terms (the equation represents the match or mismatch operation). The first equation in the previous minimum stands for the deletion operation, and the second equation represents the insertion operation.



The function $ld_{u,v}(i,j)$ thus covers all the three operations of insertion, deletion, and addition and it denotes the Levenshtein distance as measured between the first i characters for the term u and the first j characters of the term v .

There are also several interesting boundary conditions with regard to the Levenshtein edit distance:

- ❑ The minimum value that the edit distance between two terms can take is the difference in length of the two terms.
- ❑ The maximum value of the edit distance between two terms can be the length of the term that is larger.
- ❑ If the two terms are equal, the edit distance is zero.
- ❑ Hamming distance between two terms is an upper bound for Levenshtein edit distance if and only if the two terms have equal length.
- ❑ This being a distance metric also satisfies the triangle inequality property.



Example:

Let's compute the Levenshtein distance between:

String A: kitten

String B: sitting

Steps to convert "kitten" to "sitting":

1. Substitute 'k' → 's' → "sitten"
2. Substitute 'e' → 'i' → "sittin"
3. Insert 'g' at the end → "sitting"

Total edits: 3

Levenshtein distance = 3



function levenshtein_distance(char u[1..m], char v[1..n]):

for all i and j, d[i,j] will hold the Levenshtein distance between the first i characters of u and the first j characters of v, note that d has $(m+1)*(n+1)$ values

int d[0..m, 0..n]

set each element in d to zero

d[0..m, 0..n] := 0

source prefixes can be transformed into empty string by dropping all characters

for i from 1 to m:

d[i, 0] := i

target prefixes can be reached from empty source prefix by inserting every character

for j from 1 to n:

d[0, j] := j

build the edit distance matrix

for j from 1 to n:

for i from 1 to m:

if $s[i] = t[j]$:

substitutionCost := 0

else:

substitutionCost := 1

$d[i, j] := \text{minimum}(d[i-1, j] + 1, \# \text{ deletion}$

$d[i, j-1] + 1, \# \text{ insertion}$

$d[i-1, j-1] + \text{substitutionCost}) \# \text{ substitution}$

the final value of the matrix is the edit distance between the terms

return $d[m, n]$



Step 1: Initialize the Matrix

- **Rows (i):** Characters of "kitten" + empty string (k, i, t, t, e, n).
- **Columns (j):** Characters of "sitting" + empty string (s, i, t, t, i, n, g).
- **Base Cases:**
 - First row (i=0): Distance = j (insert all j characters).
 - First column (j=0): Distance = i (delete all i characters).

Initial matrix (D[i][j]):

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1							
i	2							
t	3							
t	4							
e	5							
n	6							



Step 2: Fill the Matrix Using Recurrence

For each cell $D[i][j]$:

- If $s1[i-1] == s2[j-1]$, $\text{cost} = 0$ (no edit needed).
- Else, $\text{cost} = 1$ (substitution).

• Then compute:

$$D[i][j] = \min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + \text{cost})$$

• **Arrows** indicate the chosen operation:


- \leftarrow = Insertion (from $D[i][j-1]$).
- \uparrow = Deletion (from $D[i-1][j]$).
- \nwarrow = Substitution/Match (from $D[i-1][j-1]$).



FINAL MATRIX (WITH ARROWS)

	''	s	i	t	t	i	n	g
''	0	1	2	3	4	5	6	7
k	1	1↖	2←	3←	4←	5←	6←	7←
i	2	2↑	1↖	2←	3←	4↖	5←	6←
t	3	3↑	2↑	1↖	2←	3↑	4←	5←
t	4	4↑	3↑	2↑	1↖	2↑	3↑	4←
e	5	5↑	4↑	3↑	2↑	2↖	3↑	4←
n	6	6↑	5↑	4↑	3↑	3↑	2↖	3↑
g	7	7↑	6↑	5↑	4↑	3↑	3↑	2↖

ROW 1: COMPARING 'K' (S1[0]) WITH EACH LETTER IN "SITTING"

Position	Chars Compared	Left	Top	Diagonal	Chosen	Reason	Arrow
(1,1)	k vs s	2	2	0+1=1	1	Substitution	
(1,2)	k vs i	1+1=2	3	1+1=2	2	Insertion	←
(1,3)	k vs t	2+1=3	4	2+1=3	3	Insertion	←
(1,4)	k vs t	3+1=4	5	3+1=4	4	Insertion	←
(1,5)	k vs i	4+1=5	6	4+1=5	5	Insertion	←
(1,6)	k vs n	5+1=6	7	5+1=6	6	Insertion	←
(1,7)	k vs g	6+1=7	8	6+1=7	7	Insertion	←



I

Position	Chars Compared	Left	Top	Diagonal	Chosen	Reason	Arrow
(2,1)	i vs s	3	1+1=2	1+1=2	2	Deletion	↑
(2,2)	i vs i	2+1=3	3	1+0=1	1	Match	↖
(2,3)	i vs t	1+1=2	4	2+1=3	2	Insertion	←
(2,4)	i vs t	2+1=3	5	3+1=4	3	Insertion	←
(2,5)	i vs i	3+1=4	6	4+0=4	4	Match	↖
(2,6)	i vs n	4+1=5	7	5+1=6	5	Insertion	←
(2,7)	i vs g	5+1=6	8	6+1=7	6	Insertion	←

ROW 3: COMPARING 'T' (S1[2]) WITH "SITTING"

Position	Chars Compared	Left	Top	Diagonal	Chosen	Reason	Arrow
(3,1)	t vs s	4	2+1=3	2+1=3	3	Deletion	↑
(3,2)	t vs i	3+1=4	2+1=3	2+1=3	2	Deletion	↑
(3,3)	t vs t	2+1=3	2+1=3	1+0=1	1	Match	↖
(3,4)	t vs t	1+1=2	3+1=4	2+0=2	2	Match	←
(3,5)	t vs i	2+1=3	4+1=5	3+1=4	3	Insertion	↑
(3,6)	t vs n	3+1=4	5+1=6	4+1=5	4	Insertion	←
◀ (3,7)	t vs g	4+1=5	6+1=7	5+1=6	5	Insertion	←

ROW 4: COMPARING 'T' (S1[3]) WITH "SITTING"

Position	Chars Compared	Left	Top	Diagonal	Chosen	Reason	Arrow
(4,1)	t vs s	5	3+1=4	3+1=4	4	Deletion	↑
(4,2)	t vs i	4+1=5	3+1=4	3+1=4	3	Deletion	↑
(4,3)	t vs t	3+1=4	2+1=3	2+0=2	2	Match	↑
(4,4)	t vs t	2+1=3	3+1=4	1+0=1	1	Match	↖
(4,5)	t vs i	1+1=2	4+1=5	2+1=3	2	Insertion	↑
(4,6)	t vs n	2+1=3	5+1=6	3+1=4	3	Insertion	↑
(4,7)	t vs g	3+1=4	6+1=7	4+1=5	4	Insertion	←

ROW 5: COMPARING 'E' (S1[4]) WITH "SITTING"

Position	Chars Compared	Left	Top	Diagonal	Chosen	Reason	Arrow
(5,1)	e vs s	6	4+1=5	4+1=5	5	Deletion	↑
(5,2)	e vs i	5+1=6	4+1=5	4+1=5	4	Deletion	↑
(5,3)	e vs t	4+1=5	3+1=4	3+1=4	3	Deletion	↑
(5,4)	e vs t	3+1=4	2+1=3	2+1=3	2	Deletion	↑
(5,5)	e vs i	2+1=3	3+1=4	2+1=3	2	Match	↖
(5,6)	e vs n	2+1=3	4+1=5	3+1=4	3	Deletion	↑
(5,7)	e vs g	3+1=4	5+1=6	4+1=5	4	Insertion	←

ROW 6: COMPARING 'N' (S1[5]) WITH "SITTING"

Position	Chars Compared	Left	Top	Diagonal	Chosen	Reason	Arrow
(6,1)	n vs s	7	5+1=6	5+1=6	6	Deletion	↑
(6,2)	n vs i	6+1=7	5+1=6	5+1=6	5	Deletion	↑
(6,3)	n vs t	5+1=6	4+1=5	4+1=5	4	Deletion	↑
(6,4)	n vs t	4+1=5	3+1=4	3+1=4	3	Deletion	↑
(6,5)	n vs i	3+1=4	3+1=4	2+1=3	3	Deletion	↑
(6,6)	n vs n	3+1=4	4+1=5	2+0=2	2	Match	↖
(6,7)	n vs g	2+1=3	5+1=6	3+1=4	3	Deletion	↑

STEP 3: TRACE BACK THE OPTIMAL PATH

1. $D[6][7] = 3 \uparrow \rightarrow$ Came from $D[5][7]$ (Deletion of 'n').
2. $D[5][7] = 4 \leftarrow \rightarrow$ Came from $D[5][6]$ (Insertion of 'g').
3. $D[5][6] = 3 \uparrow \rightarrow$ Came from $D[4][6]$ (Deletion of 'e').
4. $D[4][6] = 3 \uparrow \rightarrow$ Came from $D[3][6]$ (Deletion of 't').
5. $D[3][6] = 4 \leftarrow \rightarrow$ Came from $D[3][5]$ (Insertion of 'n').
6. $D[3][5] = 3 \uparrow \rightarrow$ Came from $D[2][5]$ (Deletion of 't').
7. $D[2][5] = 4 \nwarrow \rightarrow$ Came from $D[1][4]$ (Substitution 'e' \rightarrow 'i').
8. $D[1][4] = 4 \leftarrow \rightarrow$ Came from $D[1][3]$ (Insertion of 't').
9. $D[1][3] = 3 \leftarrow \rightarrow$ Came from $D[1][2]$ (Insertion of 't').
0. $D[1][2] = 2 \leftarrow \rightarrow$ Came from $D[1][1]$ (Insertion of 'i').
- ❖ 1. $D[1][1] = 1 \nwarrow \rightarrow$ Came from $D[0][0]$ (Substitution 'k' \rightarrow 's').

Optimal Edit Sequence

1. **Substitute 'k' → 's'** (kitten → sitten).
 2. **Substitute 'e' → 'i'** (sitten → sittin).
 3. **Insert 'g'** (sittin → sitting).
-

Conclusion

The **Levenshtein distance** between "**kitten**" and "**sitting**" is **3**, achieved by:

1. Substitute '**k**' → '**s**'.
2. Substitute '**e**' → '**i**'.
3. Insert '**g**' at the end.

◀ The matrix and arrows clearly show the optimal path of edits.

COSINE DISTANCE AND SIMILARITY

1. The *Cosine distance* is a metric that can be actually derived from the Cosine similarity and vice versa.
2. Considering we have two terms such that they are represented in their vectorized forms, Cosine similarity gives us the measure of the cosine of the angle between them when they are represented as non-zero positive vectors in an inner product space.
3. Thus term vectors having similar orientation will have scores closer to 1 ($\cos 0$ degree) indicating the vectors are very close to each other in the same direction (near to zero degree angle between them).
4. Term vectors having a similarity score close to 0 ($\cos 90$ degree) indicate unrelated terms with a near orthogonal angle between them.
5. Term vectors with a similarity score close to -1 ($\cos 180$ degree) indicate terms that are completely oppositely oriented to each other.



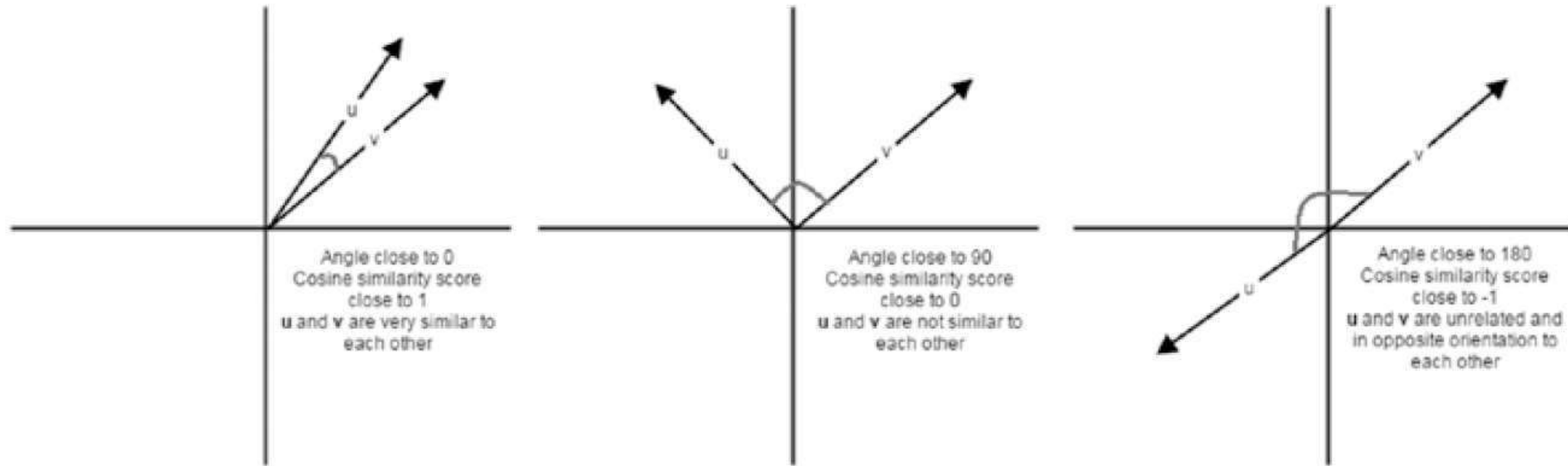


Figure . Cosine similarity representations for term vectors

where u and v are our term vectors in the vector space.

the plots show more clearly how the vectors are close or far apart from each other, and the cosine of the angle between them gives us the Cosine similarity metric.

Cosine similarity can be defined as the dot product of the two term vectors u and v , divided by the product of their L2 norms.

Mathematically, we can represent the dot product between two vectors as

$$u \cdot v = \|u\| \|v\| \cos(\theta)$$

where θ is the angle between u and v and $\|u\|$ represents the L2 norm for vector u and $\|v\|$ is the L2 norm for vector v . Thus we can derive the Cosine similarity from the above formula as

$$cs(u, v) = \cos(\theta) = \frac{u \cdot v}{\|u\| \|v\|} = \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2} \sqrt{\sum_{i=1}^n v_i^2}}$$

where $cs(u, v)$ is the Cosine similarity score between u and v . Here u_i and v_i are the various features or components of the two vectors, and the total number of these features or components is n .



we will be using the Bag of Characters vectorization to build these term vectors, and n will be the number of unique characters across the terms under analysis.

An important thing to note here is that the Cosine similarity score usually ranges from -1 to $+1$, but if we use the Bag of Characters–based character frequencies for terms or Bag of Words–based word frequencies for documents, the score will range from 0 to 1 because the frequency vectors can never be negative, and hence the angle between the two vectors cannot exceed 90 degree .

The Cosine distance is complimentary to the similarity score can be computed by the formula

$$cd(u, v) = 1 - cs(u, v) = 1 - \cos(\theta) = 1 - \frac{u \cdot v}{\|u\| \|v\|} = 1 - \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2} \sqrt{\sum_{i=1}^n v_i^2}}$$



where $cd(u, v)$ denotes the Cosine distance between the term vectors u and v . The following function implements computation of Cosine distance

EXAMPLE

- Sentence 1: "I love NLP"
- Sentence 2: "I enjoy learning NLP"

Vectors:

- $A = [1, 1, 1, 0, 0]$
- $B = [1, 0, 1, 1, 1]$

1. Dot Product:

$$1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 1 + 0 \times 1 = 1 + 0 + 1 + 0 + 0 = 2$$

2. Magnitudes:

- $\|A\| = \sqrt{1^2 + 1^2 + 1^2 + 0 + 0} = \sqrt{3} \approx 1.732$
- $\|B\| = \sqrt{1^2 + 0 + 1^2 + 1^2 + 1^2} = \sqrt{4} = 2.0$

 Cosine Similarity Formula:

$$\text{Similarity} = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

Where:

- $A \cdot B$ is the dot product
- $\|A\|$ and $\|B\|$ are magnitudes (L2 norms)

3. Cosine Similarity:

$$\frac{2}{1.732 \times 2} = \frac{2}{3.464} \approx \boxed{0.577}$$

 Cosine Distance:

$$\text{Distance} = 1 - \text{Similarity} = 1 - 0.577 \approx \boxed{0.423}$$

FUNCTION TO COMPUTE COSINE DISTANCE

```
def cosine_distance(u, v):  
    distance = 1.0 - (np.dot(u, v) /  
    (np.sqrt(sum(np.square(u))) * np.sqrt(sum(np.  
    square(v))))  
    )  
    return distance
```



ANALYZING DOCUMENT SIMILARITY

- A document is defined as a body of text which can be comprised of sentences or paragraphs of text.
- For analyzing document similarity, utils module will be used to extract features from document using the `build_feature_matrix()` function.
- Documents are vectorized using their TF-IDFs.
- After getting the vector representations of the various documents, similarity between the documents can be computed using several distance or similarity metrics which are as follows:
 - ☐ Cosine similarity
 - ☐ Hellinger-Bhattacharya distance
 - ☐ Okapi BM25 ranking



- To analyze similarity, it is assumed that the queries are in the form of three documents,
- and relevant documents for each of these three will be returned from the index of nine documents based on similarity metrics.

```
from normalization import normalize_corpus
```

```
from utils import build_feature_matrix
```

```
import numpy as np
```

```
# load the toy corpus index
```

```
toy_corpus = ['The sky is blue',
```

```
'The sky is blue and beautiful',
```

```
'Look at the bright blue sky!',
```

```
'Python is a great Programming language',
```

```
'Python and Java are popular Programming languages',
```

```
'Among Programming languages, both Python and Java are the most used in
```



Analytics',

'The fox is quicker than the lazy dog',

'The dog is smarter than the fox',

'The dog, fox and cat are good friends']

load the docs for which we will be measuring similarities

query_docs = ['The fox is definitely smarter than the dog',

'Java is a static typed programming language unlike Python',

'I love to relax under the beautiful blue sky!']



Before looking at metrics, the documents are normalized and vectorized by extracting their TF-IDF features, as shown in the following snippet:

```
# normalize and extract features from the toy corpus
```

```
norm_corpus = normalize_corpus(toy_corpus, lemmatize=True)
```

```
tfidf_vectorizer, tfidf_features = build_feature_matrix(norm_corpus,  
feature_
```

```
type='tfidf',
```

```
ngram_range=(1, 1),
```

```
min_df=0.0, max_
```

```
df=1.0)
```

```
# normalize and extract features from the query corpus
```

```
norm_query_docs = normalize_corpus(query_docs, lemmatize=True)
```

```
query_docs_tfidf = tfidf_vectorizer.transform(norm_query_docs)
```



Now we can compute the similarity between the documents



COSINE SIMILARITY

- In this method, the document vectors will be the Bag of Words model-based vectors with TF-IDF values instead of term frequencies.
- It can work with unigrams, bigrams and so on as document features during the vectorization process.
- For each of the three query documents, similarity with the nine documents is computed in toy_corpus and the n most similar documents are returned where n is a user input parameter.
- A function is defined that takes in the vectorized corpus and the document corpus for which the similarities need to be computed.
- The similarity scores will be obtained using the dot product operation as before and finally they will be sorted in reverse order to get the top n documents with the highest similarity score.




```
def compute_cosine_similarity(doc_features, corpus_features,  
top_n=3):
```

```
# get document vectors
```

```
doc_features = doc_features.toarray()[0]
```

```
corpus_features = corpus_features.toarray()
```

```
# compute similarities
```

```
similarity = np.dot(doc_features,  
corpus_features.T)
```

```
# get docs with highest similarity scores
```

```
top_docs = similarity.argsort()[::-1][:top_n]
```

```
top_docs_with_score = [(index, round(similarity[index], 3))
```

```
for index in top_docs]
```

```
return top_docs_with_score
```

corpus_features are the vectorized documents belonging to the toy_corpus index from which we want to retrieve similar documents.

These documents will be retrieved on the basis of their similarity score with doc_features, which basically represents the vectorized document belonging to each of the query_docs



```
# get Cosine similarity results for our example documents
print 'Document Similarity Analysis using Cosine Similarity'
...: print '='*60
...: for index, doc in enumerate(query_docs):
...:
...: doc_tfidf = query_docs_tfidf[index]
...: top_similar_docs = compute_cosine_similarity(doc_tfidf,
...: tfidf_features,
...: top_n=2)
...: print 'Document',index+1 ,':', doc
...: print 'Top', len(top_similar_docs), 'similar docs:'
...: print '-'*40
...: for doc_index, sim_score in top_similar_docs:
```





```
...: print 'Doc num: {} Similarity Score: {}\nDoc: {}'.  
format(doc_index+1,  
...:  
sim_score, toy_corpus[doc_index])  
...: print '-'*40  
...: print
```



OUTPUT

Document Similarity Analysis using Cosine Similarity

=====

Document 1 : The fox is definitely smarter than the dog

Top 2 similar docs:

Doc num: 8 Similarity Score: 1.0

Doc: The dog is smarter than the fox

Doc num: 7 Similarity Score: 0.426

Doc: The fox is quicker than the lazy dog

Document 2 : Java is a static typed programming language unlike Python

Top 2 similar docs:

Doc num: 5 Similarity Score: 0.837

1.0 indicates perfect similarity, 0.0 indicates no similarity, and any score between them indicates some level of similarity based on how large that score is.





Doc: Python and Java are popular Programming languages

Doc num: 6 Similarity Score: 0.661

Doc: Among Programming languages, both Python and Java are the most used in Analytics

Document 3 : I love to relax under the beautiful blue sky!

Top 2 similar docs:

Doc num: 2 Similarity Score: 1.0

Doc: The sky is blue and beautiful

Doc num: 1 Similarity Score: 0.72

Doc: The sky is blue



The output depicts the top two most relevant documents for each of the query documents based on Cosine similarity scores. Documents about *animals* are similar to the document that mentions *the fox* and *the dog*; documents about *Python* and *Java* are most similar to the query document talking about them; and *the beautiful blue sky* is indeed similar to documents that talk about *the sky* being *blue* and *beautiful*!

HELLINGER-BHATTACHARYA DISTANCE

- The Hellinger-Bhattacharya distance (HB-distance) is also called the Hellinger distance or the Bhattacharya distance.
- The Bhattacharya distance, originally introduced by A. Bhattacharya, is used to measure the similarity between two discrete or continuous probability distributions.
- E. Hellinger introduced the Hellinger integral in 1909, which is used in the computation of the Hellinger distance.
- Overall, the Hellinger-Bhattacharya distance is an f-divergence, which in the theory of probability is defined as a function $D_f(P || Q)$, which can be used to measure the difference between P and Q probability distributions.
- There are many instances of f-divergences, including KL-divergence and HB-distance.



- Note that KL-divergence is not a distance metric because it violates the symmetric condition from the four conditions necessary for a distance measure to be a metric.
- HB-distance is computable for both continuous and discrete probability distributions.
- we will be using the TF-IDF-based vectors as our document distributions. This makes it discrete distributions because we have specific TF-IDF values for specific feature terms, unlike continuous distributions.
- The Hellinger-Bhattacharya distance can be defined mathematically as

$$hbd(u, v) = \frac{1}{\sqrt{2}} \left\| \sqrt{u} - \sqrt{v} \right\|_2$$

where $hbd(u, v)$ denotes the Hellinger-Bhattacharya distance between the document vectors u and v , and it is equal to the Euclidean or L2 norm of the difference of the square root of the vectors divided by the square root of 2.





Considering the document vectors u and v to be discrete with n number of features, we can further expand the above formula into

$$hbd(u, v) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^n (\sqrt{u_i} - \sqrt{v_i})^2}$$

such that $u = (u_1, u_2, \dots, u_n)$ and $v = (v_1, v_2, \dots, v_n)$: document vectors having length n indicating n features, which are the TF-IDF weights of the various terms in the documents




```
def compute_hellinger_bhattacharya_distance(doc_features, corpus_features,
top_n=3):
    # get document vectors
    doc_features = doc_features.toarray()[0]
    corpus_features = corpus_features.toarray()

    # compute hb distances
    distance = np.hstack(
        np.sqrt(0.5 * np.sum(np.square(np.sqrt(doc_features) - np.sqrt(corpus_features)), axis=1)))

    # get docs with lowest distance scores
    top_docs = distance.argsort()[:top_n]
    top_docs_with_score = [(index, round(distance[index], 3))
        for index in top_docs]

    return top_docs_with_score
```



OKAPI BM25 RANKING

Okapi BM25 (short for *Best Matching 25*) is a popular ranking technique in information retrieval and search engines.

Originally implemented in the 1980s–90s at City University, London in the Okapi system, it's based on probabilistic relevance models developed in the 1970s–80s by researchers like S. Robertson and K. Jones.

BM25 ranks documents based on various factors, and has several variants, including **BM15**, **BM25+**, and **BM25F**.

The Okapi BM25 can be formally defined as a document ranking and retrieval function based on a Bag of Words–based model for retrieving relevant documents based on a user input query.

This query can be itself a document containing a sentence or collection of sentences, or it can even be a couple of words.



- The Okapi BM25 is actually not just a single function but is a framework consisting of a whole collection of scoring functions combined together.
- Say we have a query document QD such that $QD = (q_1, q_2, \dots, q_n)$ containing n terms or keywords and we have a corpus document CD in the corpus of documents from which we want to get the most relevant documents to the query document based on similarity scores.
- Assuming we have these, we can mathematically define the BM25 score between these two documents as

$$bm25(CD, QD) = \sum_{i=1}^n idf(q_i) \cdot \frac{f(q_i, CD) \cdot (k_1 + 1)}{f(q_i, CD) + k_1 \cdot \left(1 - b + b \cdot \frac{|CD|}{avgdl}\right)}$$

where the function $bm25(CD, QD)$ computes the BM25 rank or score of the document CD based on the query document QD . The function $idf(q_i)$ gives us the *inverse document frequency* (IDF) of the term q_i in the corpus that contains CD and from which we want to retrieve the relevant documents

$$idf(t) = 1 + \log \frac{C}{1 + df(t)}$$



where $idf(t)$ represents the idf for the term t and C represents the count of the total number of documents in our corpus and $df(t)$ represents the frequency of the number of documents in which the term t is present.

- The function $f(q_i, CD)$ gives us the frequency of the term q_i in the corpus document CD .
- The expression $|CD|$ indicates the total length of the document CD which is measured by its number of words, and the term $avgdl$ represents the average document length of the corpus from which we will be retrieving documents.
- Besides that, there are two free parameters, k_1 , which is usually in the range of $[1.2, 2.0]$, and b , which is usually taken as 0.75.
- We will be taking the value of k_1 to be 1.5 in our implementation.



COMPUTATIONAL STEPS FOR CALCULATION OF BM25 SCORES FOR DOCUMENTS

Various steps to successfully implement and compute BM25 scores for documents:

1. Build a function to get inverse document frequency (IDF) values for terms in corpus.
2. Build a function for computing BM25 scores for query document and corpus documents.
3. Get Bag of Words–based features for corpus documents and query documents.
4. Compute average length of corpus documents and IDFs of the terms in the corpus documents using function from point 1.
5. Compute BM25 scores, rank relevant documents, and fetch the n most relevant documents for each query document using the function in point 2.



FUNCTION

```
import scipy.sparse as sp  
def compute_corpus_term_idfs(corpus_features, norm_corpus):  
    dfs = np.diff(sp.csc_matrix(corpus_features, copy=True).indptr)  
    dfs = 1 + dfs # to smoothen idf later  
    total_docs = 1 + len(norm_corpus)  
    idfs = 1.0 + np.log(float(total_docs) / dfs)  
    return idfs
```



```
def compute_bm25_similarity(doc_features, corpus_features,  
corpus_doc_lengths, avg_doc_length,  
term_idfs, k1=1.5, b=0.75, top_n=3):
```

```
# get corpus bag of words features
```

```
corpus_features = corpus_features.toarray()
```

```
# convert query document features to binary features
```


```
# this is to keep a note of which terms exist per document
```

```
doc_features = doc_features.toarray()[0]
```

```
doc_features[doc_features >= 1] = 1
```

```
# compute the document idf scores for present terms
```

```
doc_idfs = doc_features * term_idfs
```

```
 # compute numerator expression in BM25 equation  
numerator_coeff = corpus_features * (k1 + 1)
```

```
numerator = np.multiply(doc_idfs, numerator_coeff)
# compute denominator expression in BM25 equation
denominator_coeff = k1 * (1 - b +
(b * (corpus_doc_lengths /
avg_doc_length)))
denominator_coeff = np.vstack(denominator_coeff)
denominator = corpus_features + denominator_coeff
# compute the BM25 score combining the above equations
bm25_scores = np.sum(np.divide(numerator,
denominator),
axis=1)
```





```
# get top n relevant docs with highest BM25 score
top_docs = bm25_scores.argsort()[::-1][:top_n]
top_docs_with_score = [(index, round(bm25_scores[index], 3))
for index in top_docs]
return top_docs_with_score
```

In simple terms, we first compute the numerator expression in the BM25 mathematical equation we specified earlier and then compute the denominator expression. Finally, we divide the numerator by the denominator to get the BM25 scores for all the corpus documents. Then we sort them in descending order and return the top n relevant documents with the highest BM25 score



DOCUMENT CLUSTERING

Document clustering or cluster analysis is an interesting area in NLP and text analytics that applies unsupervised ML concepts and techniques.

The main premise of document clustering is similar to that of document categorization, where we start with a whole corpus of documents and are tasked with segregating them into various groups based on some distinctive properties, attributes, and features of the documents.

Document classification needs pre-labeled training data to build a model and then categorize documents.

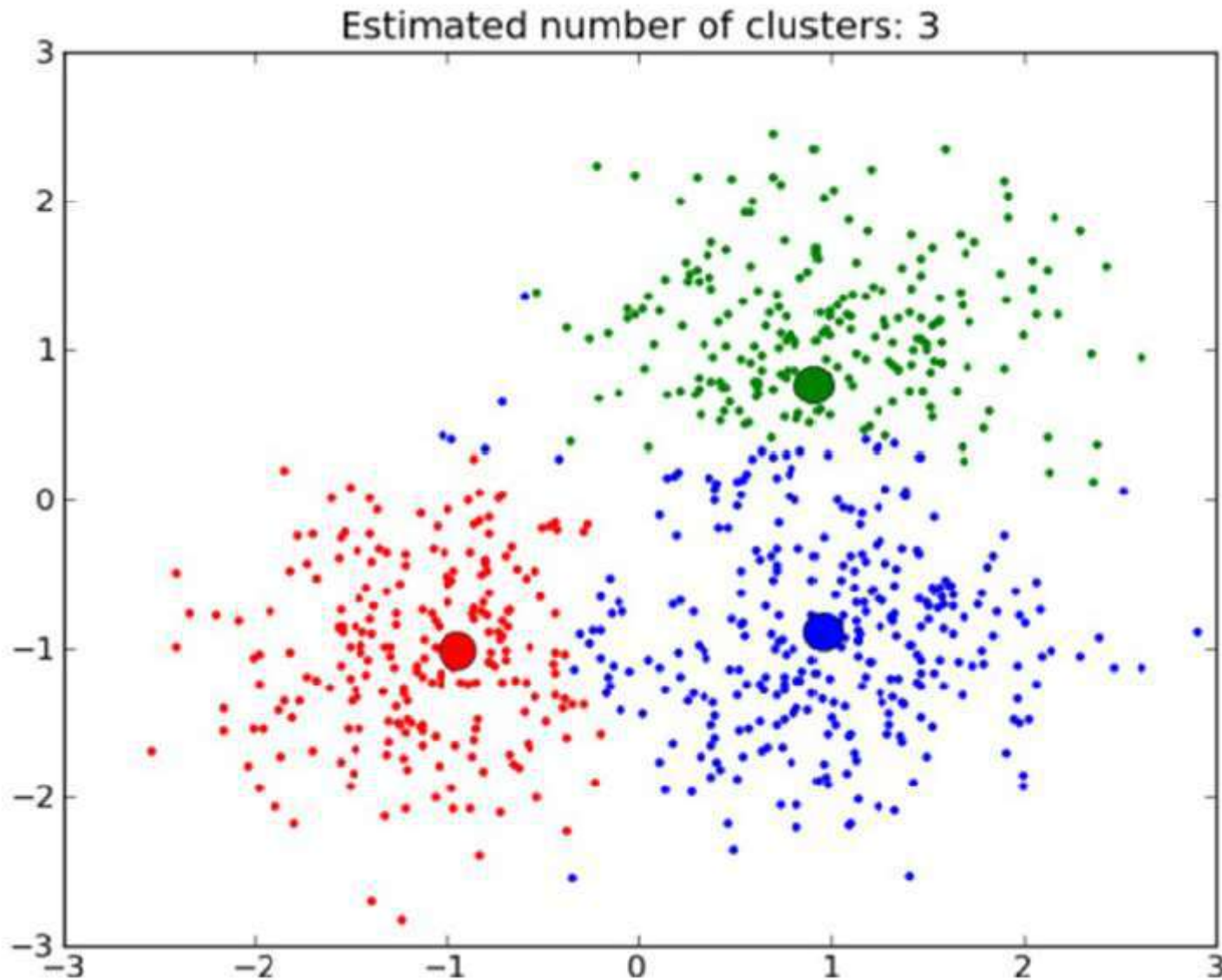
Document clustering uses unsupervised ML algorithms to group the documents into various clusters.

The properties of these clusters are such that documents inside one cluster are more similar and related to each other compared to documents belonging to other clusters.



SAMPLE CLUSTER ANALYSIS

RESULT



CLUSTERING ALGORITHMS

Some popular clustering algorithms are briefly described as follows:

➤ Hierarchical clustering models:

- These clustering models are also known as connectivity-based clustering methods and are based on the concept that similar objects will be closer to related objects in the vector space than unrelated objects, which will be farther away from them.
- Clusters are formed by connecting objects based on their distance and they can be visualized using a dendrogram.
- The output of these models is a complete, exhaustive hierarchy of clusters. They are mainly subdivided into agglomerative and divisive clustering models.



➤ Centroid-based clustering models:

- These models build clusters in such a way that each cluster has a central representative member that represents each cluster and has the features that distinguish that particular cluster from the rest.
- There are various algorithms in this, like k-means, k-medoids, and so on, where we need to set the number of clusters 'k' in advance, and distance metrics like squares of distances from each data point to the centroid need to be minimized.
- The disadvantage of these models is that you need to specify the 'k' number of clusters in advance, which may lead to local minima, and you may not get a true clustered representation of your data.

➤ Distribution-based clustering models:

- These models make use of concepts from probability distributions when clustering data points.



- The idea is that objects having similar distributions can be clustered into the same group or cluster.
- Gaussian mixture models (GMM) use algorithms like the Expectation-Maximization algorithm for building these clusters.
- Feature and attribute correlations and dependencies can also be captured using these models, but it is prone to overfitting.
- Density-based clustering models:
 - These clustering models generate clusters from data points that are grouped together at areas of high density compared to the rest of the data points, which may occur randomly across the vector space in sparsely populated areas.
 - These sparse areas are treated as noise and are used as border points to separate clusters.
 - Two popular algorithms in this area include DBSCAN and OPTICS.





three different clustering algorithms are:

- ❑ K-means clustering
- ❑ Affinity propagation
- ❑ Ward's agglomerative hierarchical clustering



K-MEANS CLUSTERING

- The k-means clustering algorithm is a centroid-based clustering model that tries to cluster data into groups or clusters of equal variance.
- The criteria or measure that this algorithm tries to minimize is inertia, also known as within-cluster sum-of-squares.
- Perhaps the one main disadvantage of this algorithm is that the number of clusters k need to be specified in advance, as is the case with all other centroid-based clustering models.
- This algorithm is perhaps the most popular clustering algorithm out there and is frequently used due to its ease of use as well as the fact that it is scalable with large amounts of data.



EXAMPLE

Word	Length (x)	Vowels (y)	Point
cats	4	1	(4, 1)
chase	5	2	(5, 2)
mice	4	2	(4, 2)
and	3	1	(3, 1)
dogs	4	1	(4, 1)
bark	4	1	(4, 1)
birds	5	1	(5, 1)
fly	3	0	(3, 0)
high	4	1	(4, 1)
and	3	1	(3, 1)
fish	4	1	(4, 1)
swim	4	1	(4, 1)
deep	4	2	(4, 2)

S1="cats", "chase", "mice", "and", "dogs", "bark",
S2="birds", "fly", "high", "and", "fish", "swim", "deep"]

Let’s use a simple way to convert each word into a 2D point:

- 1. x = word length
- 2. y = number of vowels in the word

K-Means with 3 clusters.

Let’s say the clustering ends up like this (just an example):

Cluster 1: Words like "cats", "dogs", "and" (length 3–4, 1 vowel).

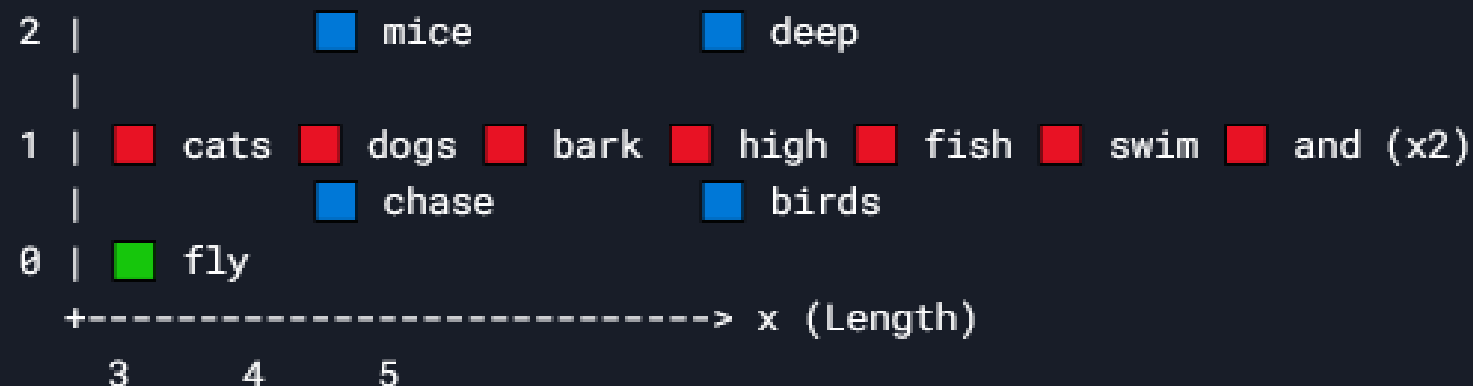
Cluster 2: Words like "chase", "mice" (length 4–5, 2 vowels).

Cluster 3: "fly" (outlier with 0 vowels).

Cluster Graph (2D Scatter Plot)

plaintext

y (Vowels)



Key:

- **Red (Cluster 1):** Short words (length 3–4) with 1 vowel.
Examples: "cats", "dogs", "and", "fish".
- **Blue (Cluster 2):** Longer words (length 4–5) or 2 vowels.
◊ *Examples:* "chase", "mice", "deep", "birds".
- **Green (Cluster 3):** Outlier ("fly") with 0 vowels.

- Consider that we have a dataset X with N data points or samples and we want to group them into K clusters where K is a user-specified parameter.
- The k-means clustering algorithm will segregate the N data points into K disjoint separate clusters C_k , and each of these clusters can be described by the means of the cluster samples.
- These means become the cluster centroids μ_k such that these centroids are not bound by the condition that they have to be actual data points from the N samples in X .
- The algorithm chooses these centroids and builds the clusters in such a way that the inertia or within-cluster sums of squares are minimized. Mathematically, this can be represented as

$$\min \sum_{i=1}^K \sum_{x_n \in C_i} \|x_n - \mu_i\|^2$$

with regard to clusters C_i and centroids μ_i such that i belongs to $\{1, 2, 3, \dots, k\}$. This optimization is an NP *hard problem*. Lloyd's algorithm is a solution to this problem.



```
from sklearn.cluster import KMeans

# define the k-means clustering function
def k_means(feature_matrix, num_clusters=5):
    km = KMeans(n_clusters=num_clusters,
                max_iter=10000)
    km.fit(feature_matrix)
    clusters = km.labels_
    return km, clusters

# set k = 5, lets say we want 5 clusters from the 100 movies
num_clusters = 5

# get clusters and assigned the cluster labels to the movies
km_obj, clusters = k_means(feature_matrix=feature_matrix,
                           num_clusters=num_clusters)
movie_data['Cluster'] = clusters
```



```
def get_cluster_data(clustering_obj, movie_data,  
feature_names, num_clusters,  
topn_features=10):  
    cluster_details = {}  
    # get cluster centroids  
    ordered_centroids = clustering_obj.cluster_centers_.argsort()[:, :-1]  
    # get key features for each cluster  
    # get movies belonging to each cluster  
    for cluster_num in range(num_clusters):  
        cluster_details[cluster_num] = {}  
        cluster_details[cluster_num]['cluster_num'] = cluster_num  
        key_features = [feature_names[index]  
for index
```



```
in ordered_centroids[cluster_num, :topn_features]]
cluster_details[cluster_num]['key_features'] = key_features
movies = movie_data[movie_data['Cluster'] == cluster_num]['Title'].
values.tolist()
cluster_details[cluster_num]['movies'] = movies
return cluster_details
```



AFFINITY PROPAGATION

- The k-means algorithm, although very popular, has the drawback that the user has to predefine the number of clusters.
- What if in reality there are more clusters or lesser clusters?
- There are some ways of checking the cluster quality and seeing what the value of the optimum k might be
- The affinity propagation (AP) algorithm is based on the concept of “message passing” among the various data points to be clustered, and no pre-assumption is needed about the number of possible clusters.
- AP creates these clusters from the data points by passing messages between pairs of data points until convergence is achieved.
- The entire dataset is then represented by a small number of exemplars that act as representatives for samples.
- These exemplars are analogous to the centroids you obtain from k-means or k-medoids.





- The messages that are sent between pairs represent how suitable one of the points might be in being the exemplar or representative of the other data point.
- This keeps getting updated in every iteration until convergence is achieved, with the final exemplars being the representatives of each cluster.
- One drawback of this method is that it is computationally intensive because messages are passed between each pair of data points across the entire dataset and can take substantial time to converge for large datasets.



STEPS INVOLVED IN THE AP ALGORITHM

Consider that we have a dataset X with n data points such that $X = \{x_1, x_2, \dots, x_n\}$, and let $\text{sim}(x, y)$ be the similarity function that quantifies the similarity between two points x and y . The AP algorithm iteratively proceeds by executing two message-passing steps as follows:

1. Responsibility updates are sent around, which can be mathematically represented as

$$r(i, k) \leftarrow \text{sim}(i, k) - \max_{k' \neq k} \{a(i, k') + \text{sim}(i, k')\}$$

where the responsibility matrix is R and $r(i, k)$ is a measure which quantifies how well x_k can serve as being the representative or exemplar for x_i in comparison to the other candidates.

2. Availability updates are then sent around which can be mathematically represented as

$$a(i, k) \leftarrow \min \left(0, r(k, k) + \sum_{i' \neq \{i, k\}} \max(0, r(i', k)) \right) \text{ for } i \neq k \text{ and}$$

availability for $i = k$ is represented as

$$a(k, k) \leftarrow \sum_{i' \neq k} \max(0, r(i', k))$$

where the availability matrix is A and $a(i, k)$ represents how appropriate it would be for x_i to pick x_k as its exemplar, considering all the other points' preference to pick x_k as an exemplar.





```
from sklearn.cluster import AffinityPropagation
def affinity_propagation(feature_matrix):
    sim = feature_matrix * feature_matrix.T
    sim = sim.todense()
    ap = AffinityPropagation()
    ap.fit(sim)
    clusters = ap.labels_
    return ap, clusters
```



```
# get clusters using affinity propagation
ap_obj, clusters = affinity_propagation(feature_matrix=feature_matrix)
movie_data['Cluster'] = clusters

# get the total number of movies per cluster
In [299]: c = Counter(clusters)
...: print c.items()
[(0, 5), (1, 6), (2, 12), (3, 6), (4, 2), (5, 7), (6, 10), (7, 7), (8, 4),
(9, 8), (10, 3), (11, 4), (12, 5), (13, 7), (14, 4), (15, 3), (16, 7)]

# get total clusters
In [300]: total_clusters = len(c)
...: print 'Total Clusters:', total_clusters
```



Total Clusters: 17

WARD'S AGGLOMERATIVE HIERARCHICAL CLUSTERING

Hierarchical Clustering creates clusters in a hierarchical tree-like structure (also called a Dendrogram). Meaning, a subset of similar data is created in a tree-like structure in which the root node corresponds to the entire data, and branches are created from the root node to form several clusters.

Hierarchical Clustering is of two types.

- **Divisive and Agglomerative Hierarchical Clustering**

Divisive Hierarchical Clustering is also termed as a top-down clustering approach. In this technique, entire data or observation is assigned to a single cluster. The cluster is further split until there is one cluster for each data or observation.

Agglomerative Hierarchical Clustering :Two most similar clusters are combined iteratively until some termination criterion is satisfied. it follows bottom up approach

EXAMPLE (WARD'S AGGLOMERATIVE HIERARCHICAL CLUSTERING)

Word	Length (x)	Vowels (y)	Point
cats	4	1	(4, 1)
chase	5	2	(5, 2)
mice	4	2	(4, 2)
and	3	1	(3, 1)
dogs	4	1	(4, 1)
bark	4	1	(4, 1)
birds	5	1	(5, 1)
fly	3	0	(3, 0)
high	4	1	(4, 1)
and	3	1	(3, 1)
fish	4	1	(4, 1)
swim	4	1	(4, 1)
deep	4	2	(4, 2)

S1="cats", "chase", "mice", "and",
"dogs", "bark",
S2="birds", "fly", "high", "and", "fish",
"swim", "deep"]

Let's use a simple way to convert each word into a 2D point:

- 1. x = word length
- 2. y = number of vowels in the word

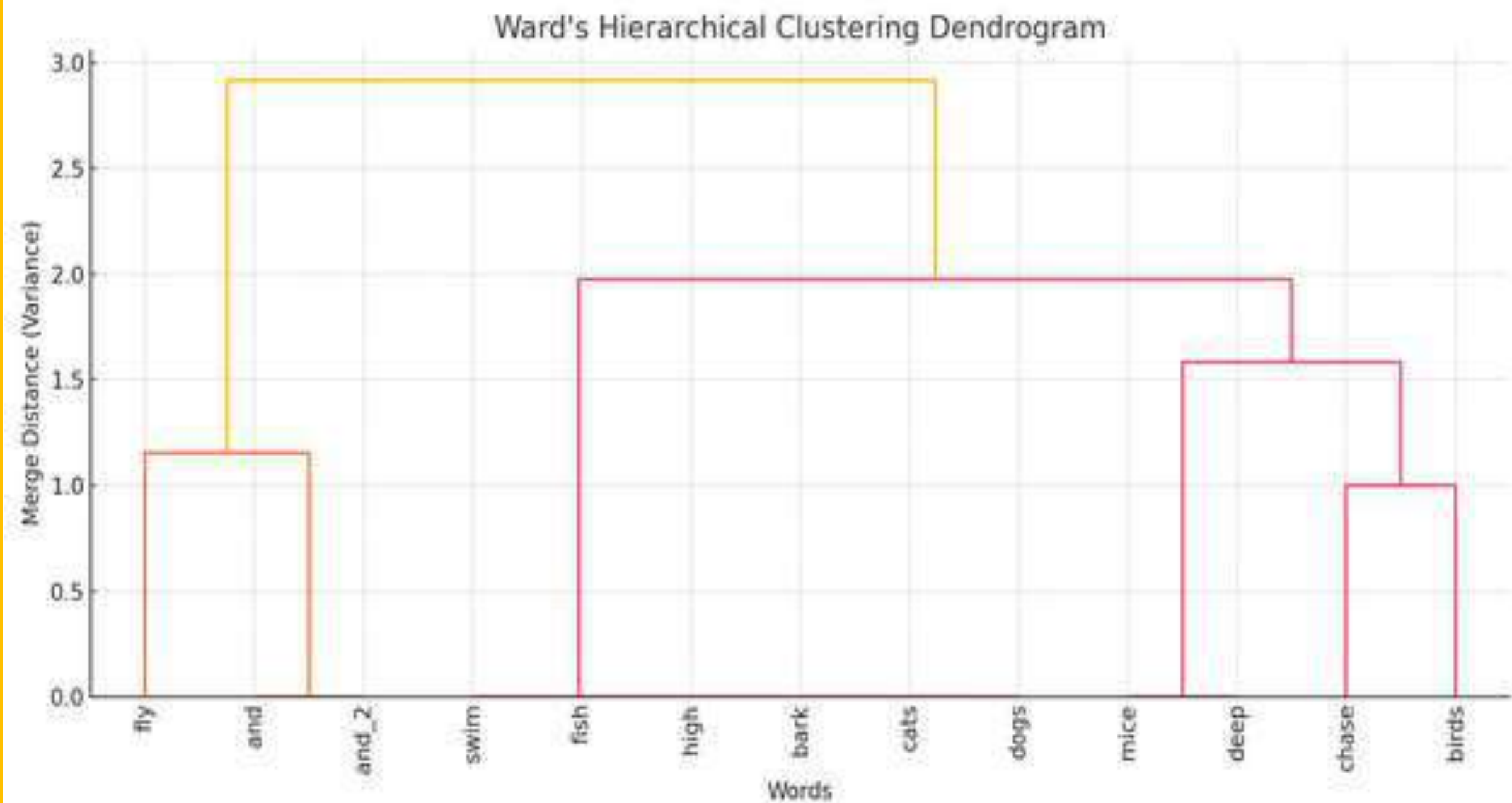
Distance between cats (4,1) and dogs (4,1)
→ $\sqrt{((4-4)^2 + (1-1)^2)} = 0$

Distance between chase (5,2) and mice (4,2)
→ $\sqrt{((5-4)^2 + (2-2)^2)} = 1$

Distance between fly (3,0) and and (3,1)
→ $\sqrt{((3-3)^2 + (0-1)^2)} = 1$

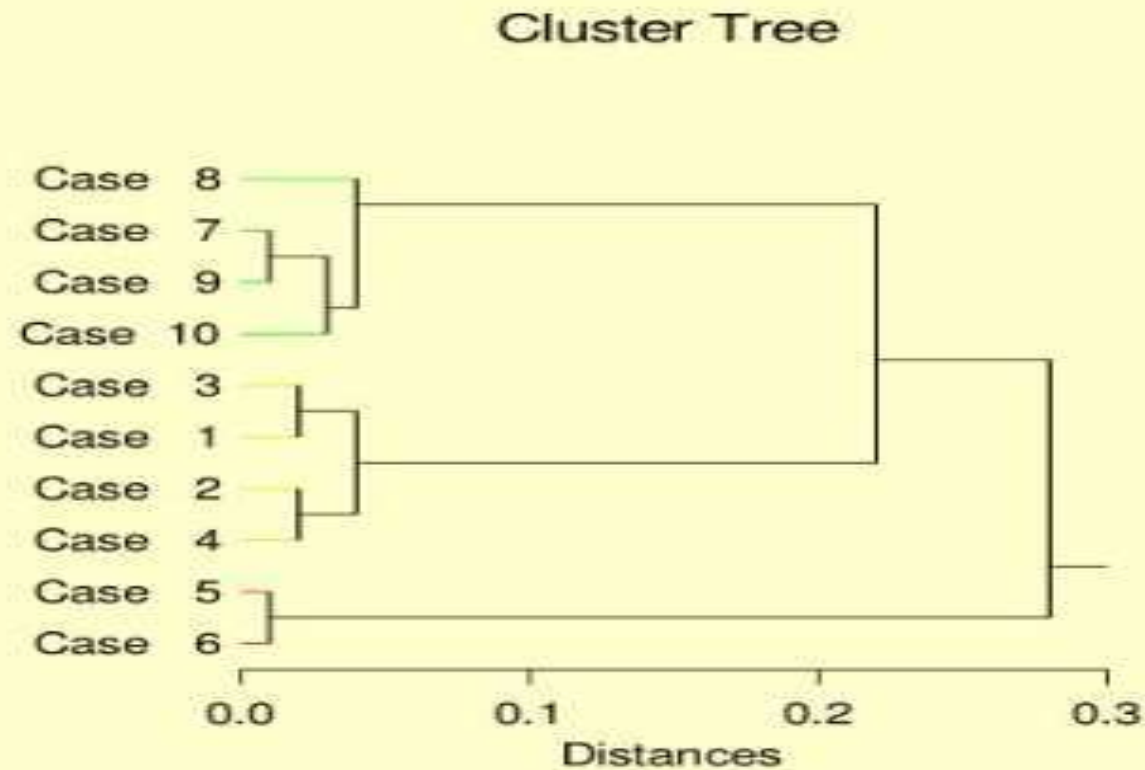
There are 78 different combinations

Let's generate the dendrogram to visualize the hierarchical clustering.



DOCUMENT CLUSTERING

- Usually represented by:
 - Dendrogram or tree-diagram



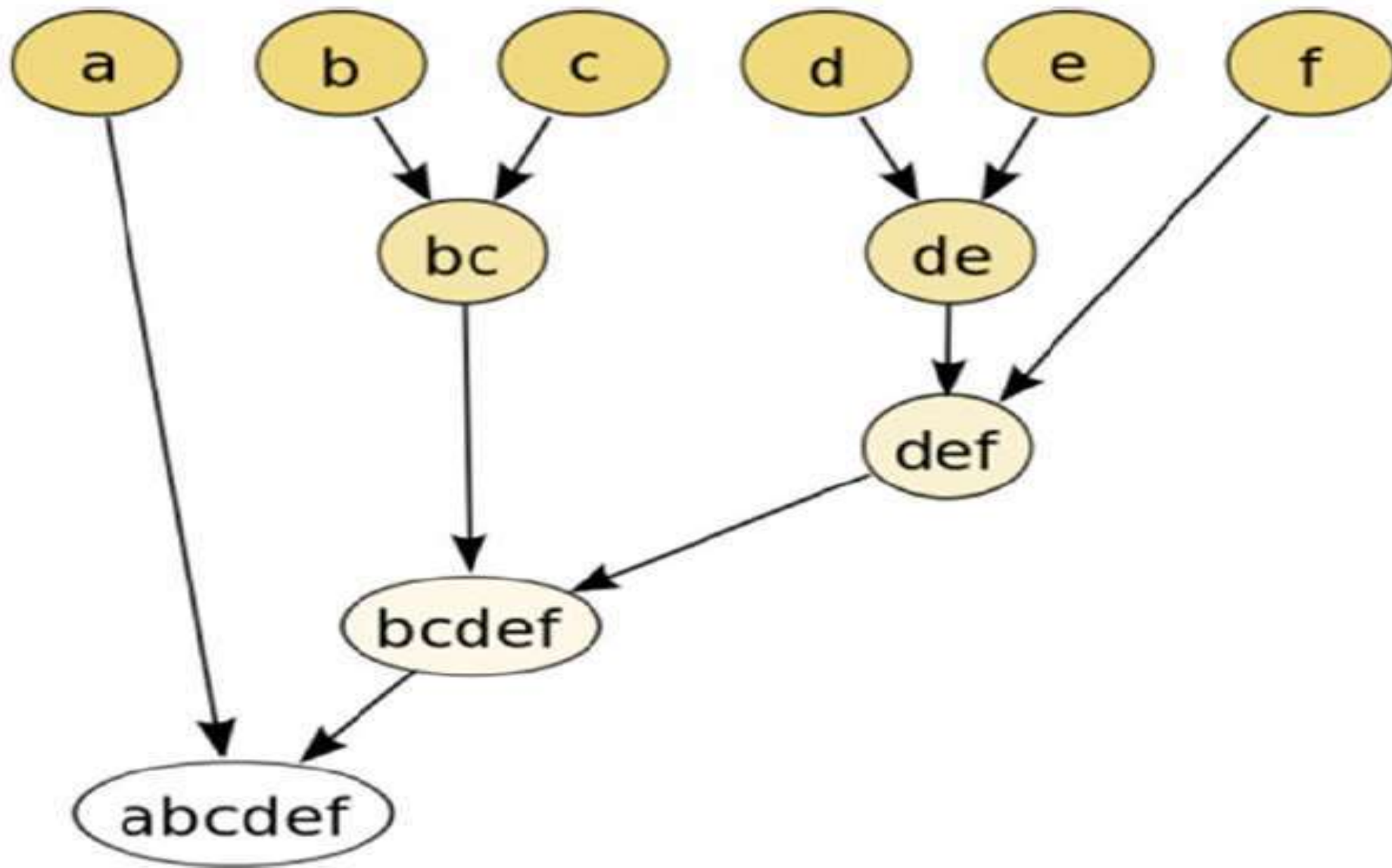


Figure 6-6. Agglomerative hierarchical clustering representation





In agglomerative clustering, for deciding which clusters we should combine when starting from the individual data point clusters, we need two things:

- A distance metric to measure the similarity or dissimilarity degree between data points. We will be using the Cosine distance/ similarity in our implementation.
- A linkage criterion that determines the metric to be used for the merging strategy of clusters. We will be using Ward's method here





```
from scipy.cluster.hierarchy import ward, dendrogram
def ward_hierarchical_clustering(feature_matrix):
    cosine_distance = 1 - cosine_similarity(feature_matrix)
    linkage_matrix = ward(cosine_distance)
    return linkage_matrix
```



```
def plot_hierarchical_clusters(linkage_matrix, movie_data, figure_
size=(8,12)):
    # set size
    fig, ax = plt.subplots(figsize=figure_size)
    movie_titles = movie_data['Title'].values.tolist()
    # plot dendrogram
    ax = dendrogram(linkage_matrix, orientation="left", labels=movie_titles)
    plt.tick_params(axis= 'x',
    which='both',
    bottom='off',
    top='off',
    labelbottom='off')
    plt.tight_layout()
    plt.savefig('ward_hierachical_clusters.png', dpi=200)
```



THANK YOU



www.reva.edu.in

