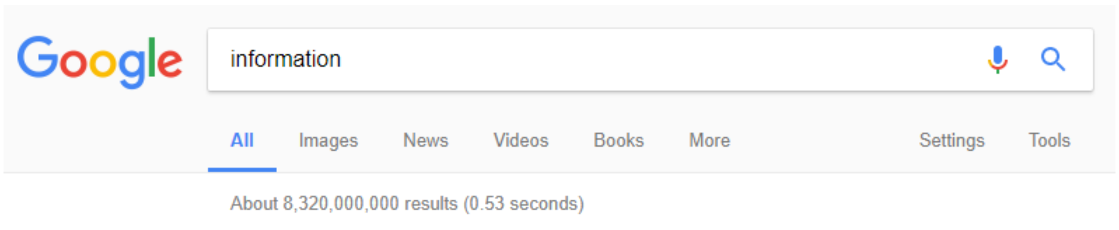
Lab 4 – Indexing, Vector Space Model

1 Indexing

Finding relevant information in a vast corpus may be a challenging task. Imagine that you want to find all documents that include the word “information” and your collection contains $1,000,000$ documents. A full-scan may take even hours to complete this task, depending on the size of documents as well as on used hardware. Such a naïve approach would work well only for a very small corpus. For large-scale collections, documents have to be indexed. An index is an auxiliary data structure that aids a search engine in searching through documents. When searching for a particular word, the index can be used, e.g., to quickly determine which documents contain such a word. Consider the search results depicted in Figure 1. Finding $8,310,000,000$ relevant documents in 0.53 seconds (January 2018) is indeed an impressive result. Without indexing, an efficient search would not be possible.



About 8,320,000,000 results (0.53 seconds)

Figure 1: Google search results.

2 Invertex index

The inverted index is a simple yet very effective data structure that allows searching for terms throughout the collection of documents. For each term (word), there is an associated list indicating documents containing this word. Each document is represented by a unique serial number *docID* (as with documents, terms can be represented by *termID*). For instance “information $\rightarrow [1 \rightarrow 5 \rightarrow 10]$” means that the term “information” is contained in the 1^{st}, 5^{th}, and in the 10^{th} document. The list is referred to as a posting list. When building the inverted index, some additional data may be stored, e.g., a position of a term (such an index is referred to as a full inverted index). When using index structures, data access is much faster at the cost of additional memory. Figure 2 depicts an example process of building an inverted index.

D1 = "new home sales top forecast

D2 = "home sales rise in july home"

D3 = "increase in home sales in july"

D4 = "july new home sales rise"

(a) data collection

term	docID
new	1
home	1
sales	1
top	1
forecast	1
home	2
sales	2
rise	2
in	2
july	2
home	2
increase	3
in	3
home	3
sales	3
in	3
july	3
july	4
new	4
home	4
sales	4
rise	4

(b) sort & merge

term	docID
forecast	1
home	1
home	2
home	2
home	3
home	4
in	2
in	3
in	3
increase	3
july	2
july	3
july	4
new	1
new	4
rise	2
rise	4
sales	1
sales	2
sales	3
sales	4
top	1

(c) Inverted index

term	posting list
forecast	[1]
home	[1 → 2 → 3 → 4]
in	[2 → 3]
increase	[3]
july	[2 → 3 → 4]
new	[1 → 4]
rise	[2 → 4]
sales	[1 → 2 → 3 → 4]
top	[1]

(d) Full inverted index

term	posting list
forecast	[1:[5]]
home	[1:[2]] → 2:[1,6] → 3:[3] → 4:[3]]
in	[2:[4]] → 3:[2,5]]
increase	[3:[1]]
july	[2:[5] → 3:[6]] → 4:[1]]
new	[1:[1] → 4:[2]]
rise	[2:[3] → 4:[5]]
sales	[1:[3] → 2:[2] → 3:[4] → 4:[4]]
top	[1:[4]]

Figure 2: Building an inverted index.

3 Suffix trie (tree)

Suffix trie and tree are trees (data structures) that contain all suffixes of a word, a sentence, or even a whole text. Different paths $root \to leaf$ represent different suffixes. The difference between a suffix trie and a suffix tree is that the latter is optimized and does not have redundant nodes. The example suffix tree is depicted in Figure 3. For a given string S of length m, a suffix tree T has the following properties:

- T is a rooted directed tree: one root, m leaves, directed edges (Figure 3: 7 leaves).
- Each edge is labelled with a non-empty substring of S (Figure 3: e.g., "A", "NA", "BANANA").
- Concatenation of labels on the path $root \to i^{th}\ leaf$ constitutes a suffix which starts with i^{th} letter (Figure 3: e.g., "NA"+ "NA\$" = "NANA\$" – 3^{rd} suffix).
- Each non-leaf, non-root node (internal node) has at least two children.

- Labels of outgoing edges must start with different letters (Figure 3: Root $\rightarrow$ "A", "BANANA", "NA").
- A special sign (Figure 3: \$) is used to indicate the end of the S word.

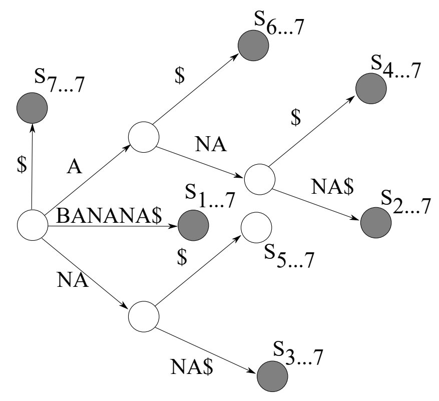


Figure 3: Suffix tree for the text "BANANA".

Why use suffix trees? Because they allow processing string-based queries efficiently. For instance:

- Verifying if a string P of length n is a substring of S in $O(n)$.
- Finding the longest common substring of two strings s_1 and s_2 in $O(n_1 + n_2)$.

How to build a suffix tree? Let $S_{i...j}$ be a substring of S, starting from the i^{th} letter and ending on the j^{th} letter.

3.1 Naïve approach

- $O(m^2)$
- Off-line algorithm.

Algorithm 1 Naïve algorithm for building a suffix tree

```
1: for $i = m$ to 1 do
2:     $P$ = find the longest prefix of $S_{i...m}$ in $T$
3:     Split the last node at $P$ and add a new edge labelled with remaining letters
```

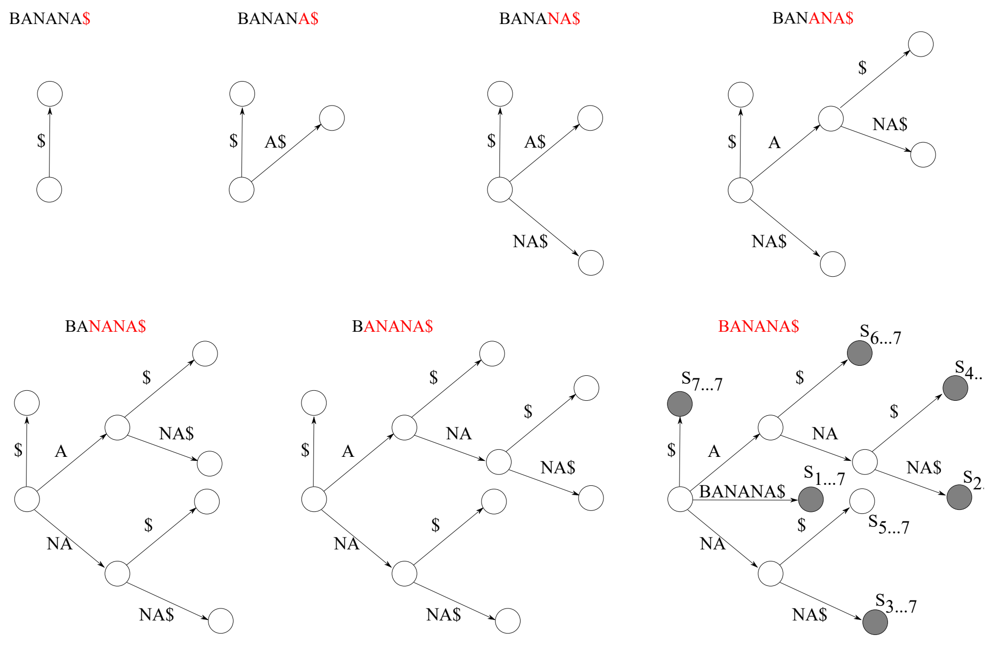


Figure 4: Naïve algorithm for building a suffix tree for a word "BANANA$".

3.2 Ukkonen's algorithm

- Two implementations: basic $O(m^3)$ and fast $O(m)$.
- On-line algorithm.

Algorithm 2 Ukkonen's algorithm for building a suffix tree (S_i – i^{th} letter of S)

```
1: Add the first letter to $T$
2: for $i = 1$ to $m - 1$ do
3:     for $j = 1$ to $i + 1$ do
4:         $P$ = find a path $S_{j...i}$ starting from the root.
5:         if $P$ ends with a leaf (see Rule 1 in Figure 5) then
6:             Add $S_{i+1}$ to $P$
7:         else if $P$ does not end with a leaf and $S_{i+1}$ does not appear after $P$ (see Rule 2 in Figure 5)
   then
8:             Construct a new leaf by adding $S_{i+1}$ to $S_{j...i}$ (split the edge)
9:         else if $P$ does not end with a leaf and $S_{i+1}$ appears after $P$ (see Rule 3 in Figure 5) then
10:            Do nothing :)
```

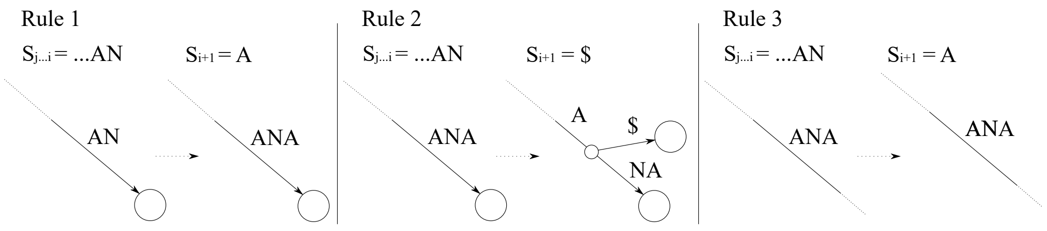


Figure 5: Three rules.

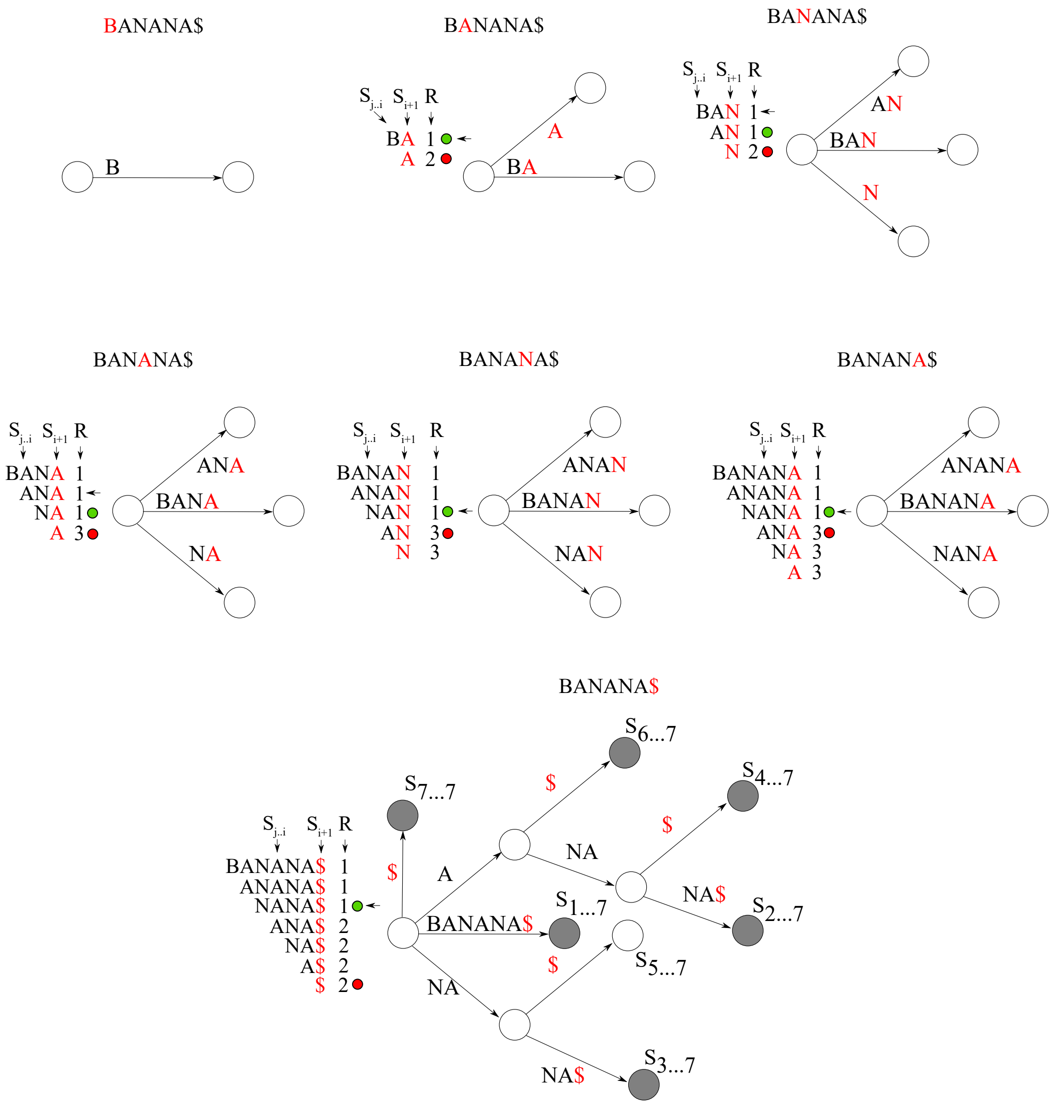
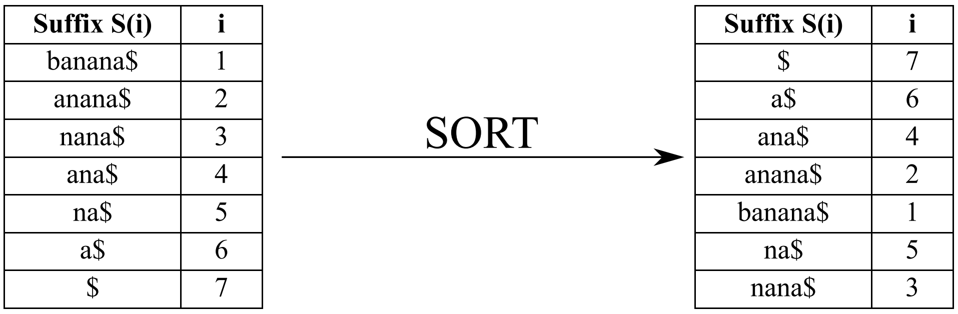


Figure 6: Ukkonen's algorithm for building a suffix tree for a word "BANANA$".

4 Suffix array

Assume that $S(i)$ is an i^{th} suffix of a word S. Suffix array is an array of integers being ordered such that $S(i)$ are sorted lexicographically (see Figure 7). Suffix arrays were proposed as an alternative to suffix trees, however, a size of an array (in bytes) may exceed the size of the original text.



Suffix S(i)	i
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

Suffix S(i)	i
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

Figure 7: Suffix array for a word “BANANA$”.

Why use suffix arrays? As with suffix trees, suffix arrays allow processing string-based queries efficiently. They can be used, e.g., to find all occurrences of P (substring) in S efficiently. The idea is to find all suffixes $S(i)$ such that P is their prefix. All $S(i)$ that match P are clustered together as the suffixes are sorted lexicographically. One may use a binary search to find the first and the last matching $S(i)$.

4.1 Qsufsort algorithm (Larsson, Sadakane)

Algorithm 3 Qsufsort algorithm for building a suffix array

```
1: Create an array $I$ which contains sorted indexes (suffixes). Initially, sort all characters of $S$ ($ is
   before $a$). Then, let $I[i]$ (for $i = 1, \ldots, m$) be an index of a suffix at $i^{th}$ position (after sort). E.g.
   (see Figure 8), $7^{th}$ suffix (“$”) should be at the first position. Thus, $I[1] = 7$.
2: Set $h = 1$.
3: Compute the worst possible position $V[i]$ each $I[i]$-$th$ suffix can attain. E.g. (see Figure 8), $S(2)$,
   $S(4)$ and $S(6)$ starts with “a” and the worst possible position is 4 ($\$, a, a, a, \ldots$). Two groups can
   be observed: letter “a” (3 suffixes) and letter “n” (2 suffixes).
4: If case of ambiguity, i.e., when $V[i]$ are not unique, the suffixes have to be sorted. For each $I[i]$-$th$
   suffix in a group derive the worst position of the $I[i] + h$ suffix, i.e., V’[i] = V[j] such that I[j] =
   I[i]+h.
5: Sort I (only in groups) with respect to V’.
6: If there are still some groups in V’, multiply h by 2 and return to 3.
```

		1	2	3	4	5	6	7	
S	=[	**b**	**a**	**n**	**a**	**n**	**a**	**$**	]
I[i]	=[	7	2	4	6	1	3	5	]
V[i]	=[	1	4	4	4	5	7	7	]
V’[i]= V[I[i] + 1]	=[		7	7	1		4	4	]
I[i]	=[								]
V[I[i]]	=[								]
V’[i] = V[I[i] + 2]	=[								]
I[i]	=[								]

		1	2	3	4	5	6	7	
S	=[	**b**	**a**	**n**	**a**	**n**	**a**	**$**	]
I[i]	=[	7	2	4	6	1	3	5	]
V[i]	=[	1	4	4	4	5	7	7	]
V’[i] = V[I[i] + 1]	=[		7	7	1		4	4	]
I[i]	=[	7	6	2	4	1	3	5	]
V[I[i]]	=[	1	2	4	4	5	7	7	]
V’[i] = V[I[i] + 2]	=[			**4**	**2**		**7**	**1**	]
I[i]	=[	**7**	**6**	**4**	**2**	**1**	**5**	**3**	]

Figure 8: Qsufsort algorithm for building a suffix array for a word “BANANA$”.

5 Vector space model and similarity

5.1 Vector space model

The vector space model represents documents as vectors. Having a dictionary of terms, each term corresponds to a specific component of a vector. Each component may be perceived as an axis in multidimensional space. Coordinate-values of a vector may be, e.g., weights. The basic representations, also known as the bag-of-words representations, are:

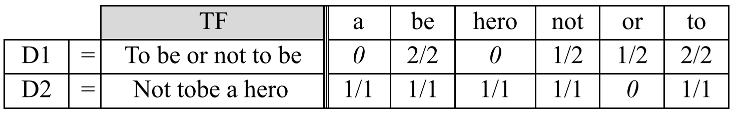
1. **Binary representation:** each element is simply a 0/1 (true/false) flag that indicates whether a particular term occurs in a document or not. E.g.:

		Binary	a	be	hero	not	or	to
D1	=	To be or not to be	0	1	0	1	1	1
D2	=	Not to be a hero	1	1	1	1	0	1

2. Then, one may extend the binary model by specifying a number of occurrences of a term. E.g.,:

			a	be	hero	not	or	to
D1	=	To be or not to be	0	2	0	1	1	2
D2	=	Not to be a hero	1	1	1	1	0	1

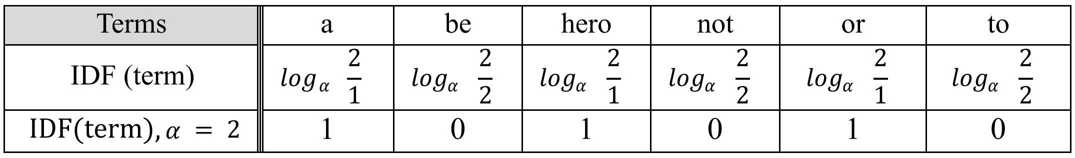
3. **Term frequency (TF)**: The number of occurrences is normalised. Usually, it is divided by the the number of occurrences of the most frequent term in the document. E.g.,:



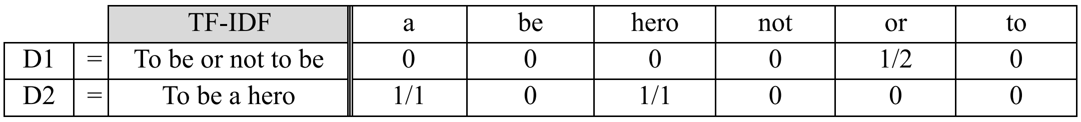
		TF	a	be	hero	not	or	to
D1	=	To be or not to be	*0*	2/2	*0*	1/2	1/2	2/2
D2	=	Not tobe a hero	1/1	1/1	1/1	1/1	*0*	1/1

4. **TF + Inverted document frequency (TF-IDF)**:

   Unlike TF, IDF is a global coefficient taking into account all documents when computing an IDF vector for a single document. IDF value for a given term is based on the number of documents which contain this term. Let D_i be the number of documents which contain an i^{th} term. Then, let $|D|$ be the total number of documents in the collection. $IDF(i)$ is defined as: $log_\alpha(|D|/D_i)$. The greater is D_i, the lower is $IDF(i)$. The goal of IDF is to decrease the importance of terms occurring in many documents and increase the importance of unique terms. IDF can be combined with TDF, using the following formula: $TF\text{-}IDF(document, i^{th}$ $term) = TF(document, i^{th}) \cdot IDF(i^{th})$:



Terms	a	be	hero	not	or	to
IDF (term)	$log_\alpha \frac{2}{1}$	$log_\alpha \frac{2}{2}$	$log_\alpha \frac{2}{1}$	$log_\alpha \frac{2}{2}$	$log_\alpha \frac{2}{1}$	$log_\alpha \frac{2}{2}$
IDF(term), $\alpha = 2$	1	0	1	0	1	0

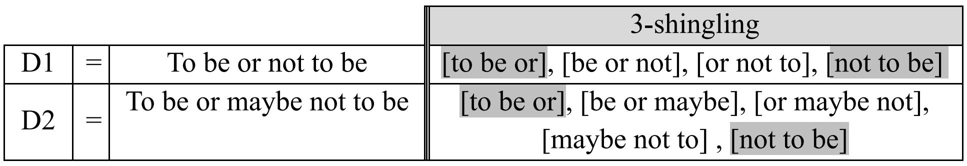


		TF-IDF	a	be	hero	not	or	to
D1	=	To be or not to be	0	0	0	0	1/2	0
D2	=	To be a hero	1/1	0	1/1	0	0	0

Some other representations

1. **Full representation**: Extends standard bag-of-words representations by incorporating term locations.

2. **w-shingle** is a set of all unique "shingles" (n-grams) of a document. An *n-gram* ($w = n$) is a vector containing n terms which occur consecutively in a document. w-shingle is usually used to detect a plagiarism. E.g.:



			3-shingling
D1	=	To be or not to be	[to be or], [be or not], [or not to], [not to be]
D2	=	To be or maybe not to be	[to be or], [be or maybe], [or maybe not], [maybe not to] , [not to be]

6 Similarity

To select documents which are relevant to user's query q, similarity between the query and documents must be computed.

6.1 Cosine similarity

When using a vector space model, one may perceive q as a document. Thus, q can be represented as a vector. To compare two vectors, one may compute a cosine similarity, which is the cosine of an angle between two vectors, defined as:

$$cos(\overrightarrow{a}, \overrightarrow{b}) = \frac{\overrightarrow{a} \cdot \overrightarrow{b}}{|\overrightarrow{a}| \cdot |\overrightarrow{b}|} = \frac{\sum_i(\overrightarrow{a_i} \cdot \overrightarrow{b_i})}{\sqrt{\sum_i(\overrightarrow{a_i}^2) \sum_i(\overrightarrow{b_i}^2)}}.$$

Consider a dictionary consisting of two terms. An example query q and documents D_1 and D_2 are illustrated in 2-dimensional vector space (see Figure 7). One may compute a cosine of an angle between D_1 and q as well as D_2 and q and compare the results. Clearly, D_2 is more similar to q (the angle is lesser/cosine value is greater).

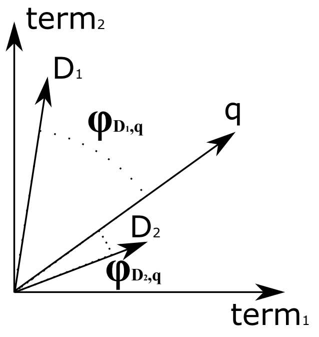


Figure 9: Cosine similarity.

6.2 Jaccard index

Jaccard index (or a coefficient) is used for comparing the similarity and diversity of two sets. It is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

Jaccard index may be used to compare two documents if they are represented by sets. For instance:

- A document bay be represented by a set of unique terms it contains (special case of 1-*shingling*). E.g.,

			Terms (1-shingles)
q	=	I like carrots	I, like, carrots
D1	=	I like carrots very much	I, like, carrots, very, much
D2	=	I prefer carrots	I, prefer, carrots

	q	D1	D2
q	1	3/5	2/4
D1		1	2/6
D2			1

Figure 10: Jaccard inddex values.

- A document may be represented by a set of w-shingles. E.g.,

			3-shingles
q	=	I like carrots	[I like carrots]
D1	=	I like carrots very much	[I like carrots], [like carrots very], [carrots very much]
D2	=	I prefer carrots	[I prefer carrots]

	q	D1	D2
q	1	1/3	0/2
D1		1	0/2
D2			1

Figure 11: Jaccard inddex values.