UiT The Arctic University of Norway

DTE-2501

Natural Language Processing

Levenshtein Distance & Term-Frequency Inverse Document Frequency

Andreas Dyrøy Jansson – Phd Candidate, UiT Narvik

Room: C3190

Email: Andreas.d.jansson@uit.no

Shayan Dadman – Phd Candidate, UiT Narvik

Room: D3430

Email: Shayan.dadman@uit.no

More practical applications of NLP

Levenshtein distance for spell checking

Inverse document frequency for keyword extraction

Levenshtein distance I

- A way to measure the «edit distance» between two strings
  - I.e., how many changes do we have to make to go from one string to the other
  - Insertions, deletions and substitutions

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}\big(\operatorname{tail}(a), b\big) \\ \operatorname{lev}\big(a, \operatorname{tail}(b)\big) \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) \end{cases} & \text{otherwise,} \end{cases}$$

Levenshtein distance II

- Example: change *survey* to *surgery*
    1. survey -> surgey (substitution of v for g)
    2. surgey -> surgery (insertion of r)
    - Levenshtein distance = 2

- Example 2: change *robot* to *rabbit*
    1. robot -> rabot (substitution of o for a)
    2. rabot -> rabbot (insertion of b)
    3. rabbot -> rabbit (substitution of o for i)
    - Levenshtein distance = 3

Levenshtein distance III

- Can be computed using (tabular) dynamic programming
  - Wagner–Fischer algorithm
- Given two strings, *a* and *b*:
  - Create a matrix of size *len(a)* + 1 by *len(b)* + 1
  - Initialize the matrix with worst-case values
    - I.e., how many operations must we do to create the strings from nothing?
    - “” to “” requires 0 changes, “” to “s” = 1, “” to “su” = 2, “” to “sur” = 3
  - We are interested in the number of changes to go from *a* to *b*
    - *“sur” to “sur”* requires 0 changes, “surg” to “surv” is 1 change etc.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1							
u	2							
r	3							
v	4							
e	5							
y	6							

Wagner–Fischer algorithm

- Given *a* = “survey”, *b* = “surgery”
- Initialize *d[len(a) + 1][len(b) + 1]*
- Set row 0 and column 0 to worst-case
- FOR every character *c* in *b (j)*:
  - FOR every character *k* in *a (i)*:
    - IF *c* = *k:*
      - cost is 0
    - ELSE:
      - cost is 1
    - Set *d[i][j] = MIN(d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1]+cost)*
- The answer is 2 and is found in <u>*d[len(a)][len(b)]*</u>

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1							
u	2							
r	3							
v	4							
e	5							
y	6							

```
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 0, 1, 2, 3, 4, 5, 6]
[2, 1, 0, 1, 2, 3, 4, 5]
[3, 2, 1, 0, 1, 2, 3, 4]
[4, 3, 2, 1, 1, 2, 3, 4]
[5, 4, 3, 2, 2, 1, 2, 3]
[6, 5, 4, 3, 3, 2, 2, 2]
```

Tabular approach

	""	s	u	r	g	e	r	y
""	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

"s" to "" is 1 change
"s" to "s" is 0 changes
"s" to "su" is 1 change
"s" to "sur" is 2 changes
"s" to "surg" is 3 changes
"s" to "surge" is 4 changes
"s" to "surger" is 5 changes
"s" to "surgery" is 6 changes

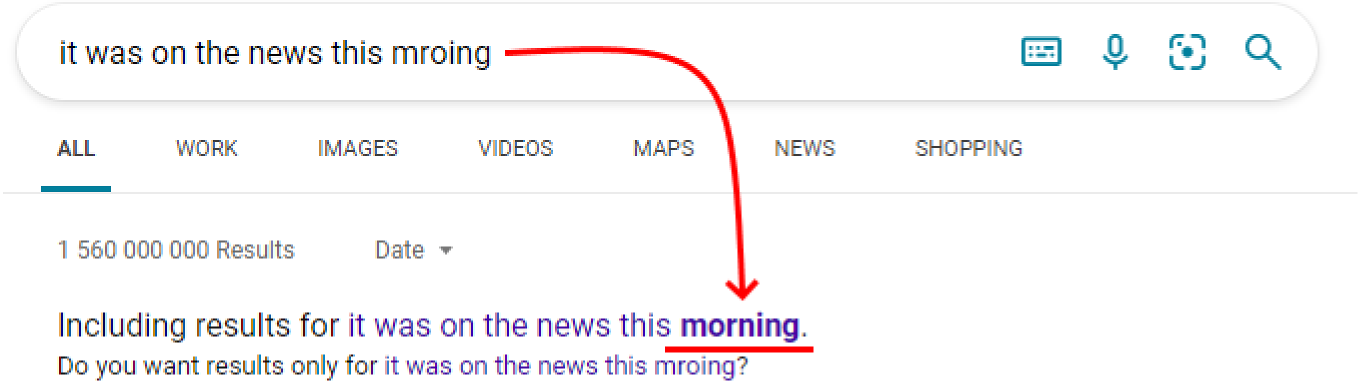
"survey" to "" is 6 changes
"survey" to "s" is 5 changes
"survey" to "su" is 4 changes
"survey" to "sur" is 3 changes
"survey" to "surg" is 3 changes
"survey" to "surge" is 2 changes
"survey" to "surger" is 2 changes
"survey" to "surgery" is 2 changes

Spell checking

- Levenshtein distance can be used for simple spell checking
  - Compare input with known words
  - Suggest word(s) with shortest edit distance
- Example:
  - Dictionary: {halloween, halo, hello, help, home}, {work, world, worth}
  - Input: hlllo wrold
  - Compare input to each known word
    - hlllo [> halloween = 5, > halo = 2, > hello = 1, > help = 3, > home = 4]
    - wrold [>work = 3, > world = 2, > worth = 4]
      - We thus conclude that «hello world» is correct and suggest it to the user

Other practical applications…

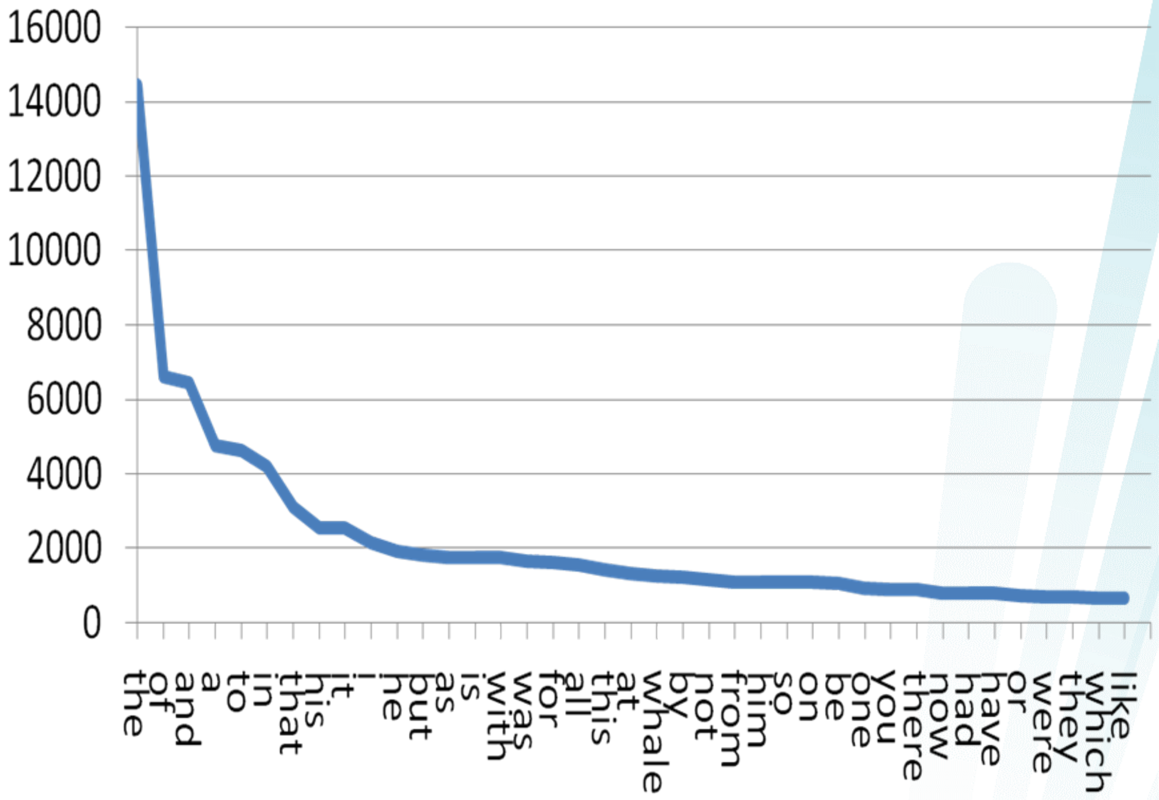
- Levenshtein distance can also be used for clustering words based on similarity, like any other distance metric
  - E.g., Euclidean distance etc. in K-NN
- Comparing two languages and their intelligibility (linguistics)
  - *lev*(“arbeidsplass”, “Arbeitsplatz”) = 3
    lev(“arbeidsplass”, “lieu de travail”) = 13
- Search engines



```python
def levenshtein_distance(s, t):
    m = len(s)
    n = len(t)

    d = [0] * (m + 1)
    for z in range (m + 1):
        d[z] = [0] * (n + 1)

    for i in range (1, m + 1):
        d[i][0] = i

    for j in range (1, n + 1):
        d[0][j] = j

    for j in range(1, n +  1):
        for i in range(1, m + 1):
            cost = 1
            if s[i - 1] == t[j - 1]:
                cost = 0
            d[i][j] = min([d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + cost])

    return d[m][n]
```

Keyword extraction and information gain

- Inverse document frequency
  - How often does a word appear in a set of texts?
  - May be used for preprocessing to remove «filler words» automatically
  - Based on Zipf's law
    - This means that the least used words are the most important, or informative
    - Words like «the, of, and, a, to» and so on add little or no value when looking for meaning in a text
  - Similarly, if a word is used multiple times in different, unrelated texts, we assume that this word is less important



Term frequency–inverse document frequency (TF-IDF) I

- Statistic for determining the importance of a word in a document
- Based on the number of occurrences per document, compared to all the words in the entire collection of documents (corpus)
- Recall Zipf's law – the less frequent a word is, the more important it is
- Term frequency TF is the relative frequency of a word in a single document

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \Rightarrow tf(t,d) = \frac{number\ of\ times\ word\ t\ appears\ in\ d}{total\ number\ of\ words\ in\ d}$$

Term frequency–inverse document frequency (TF-IDF) II

- Inverse document frequency IDF is a measure of how much information the word provides
  - If a term *t* appears in multiple, or even all, documents, we can assume that this word is less important for the meaning of a single document *d*
  - Again, based on Zipf's theory

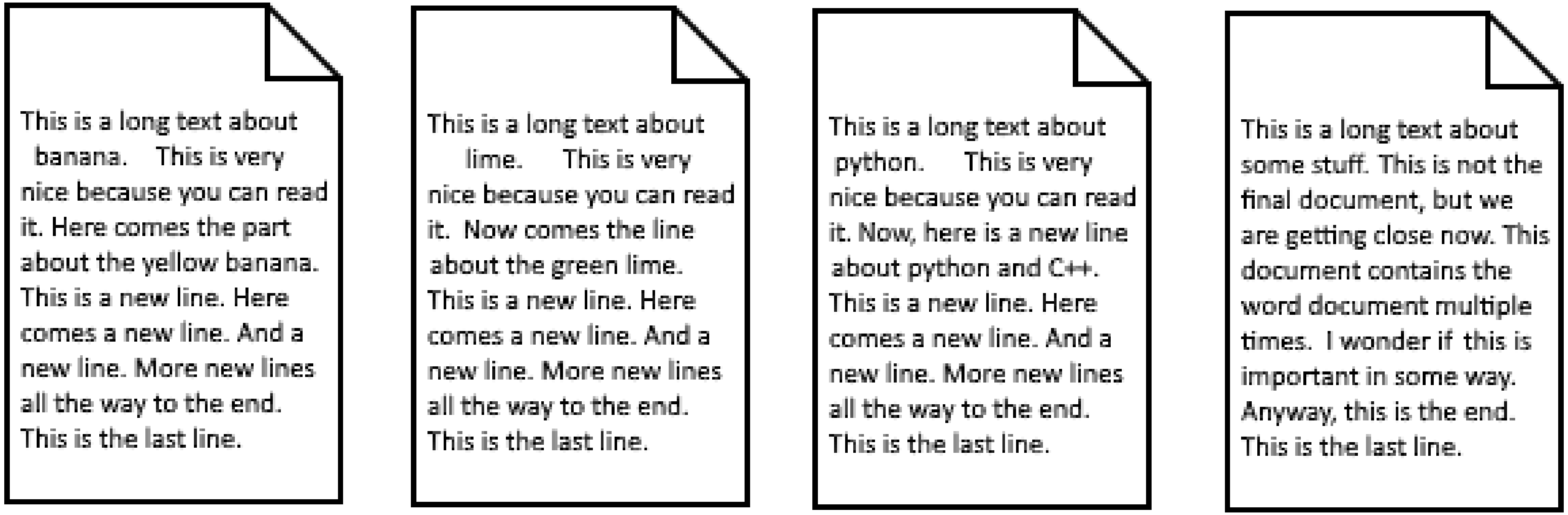
$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} \Rightarrow idf(t, D) = log\left(\frac{total\ number\ of\ documents}{number\ of\ documents\ containing\ t}\right)$$

- Combined with TF to form TF-IDF:

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

An example:

- Corpus – a set of documents containing some text:

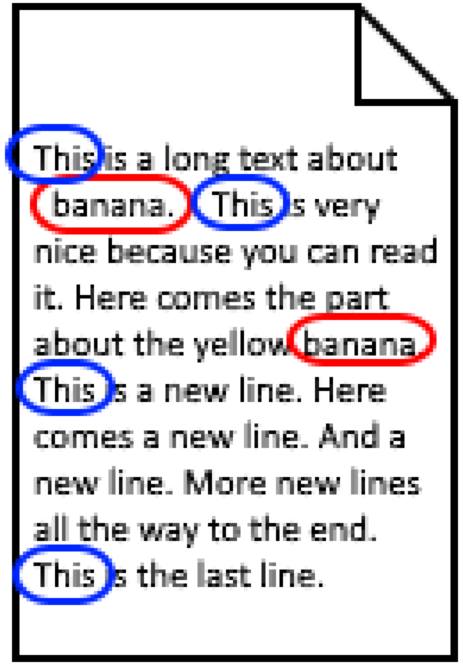


The basic concept

- We want to find the words that appear often in ONE document – these are probably important to the meaning of THAT document
  - This is the TF-part
- BUT – we want to «filter out» the words that appear often in ALL the documents
  - These do not add any value, and are most likely filler words
  - This is the IDF-part

Term Frequency

- Count how often every word appears, in each document
- Considering the first document:
  - «banana» only appears 2 times. This means that the Term Frequency is TF = 2/52 = 0.038
  - «this» appears 4 times, TF = 4/52 = 0.077
    - Is it more important to the meaning though?



Inverse Document Frequency

- This is the LOG-10 of the number of documents in our corpus (4) divided by the number of documents containing the term (word)
- «banana» only appears in the first document
  - IDF = log(4/1) = 0.6
  - TF-IDF = TF * IDF = 0.038*0.6 = 0.023
- «this» appears in all 4 documents
  - IDF = log(4/4) = log(1) = 0
  - TF-IDF = TF * IDF = 0.077 * 0 = 0
  - We can conclude that «this» is a *stop word*

OO-Implementation

- A Term-class
  - Holds the word itself, number of occurrences in its document, and the TF-iDF value
  - Comparator functions for sorting
  - A function to compute the TF-IDF

```python
class Term:
    def __init__(self, word):
        self.word = word
        self.count = 1
        self.tfidf = 0

    def increase_count(self):
        self.count += 1

    def __lt__(self, other):
        return self.tfidf < other.tfidf

    def __gt__(self, other):
        return self.tfidf > other.tfidf

    def __eq__(self, other):
        return self.tfidf == other.tfidf

    def get_word(self):
        return self.word

    def compute_tfidf(self, word_count, N, num_occ):
        self.tfidf = (self.count / word_count) \
            * (math.log(N/num_occ))
```

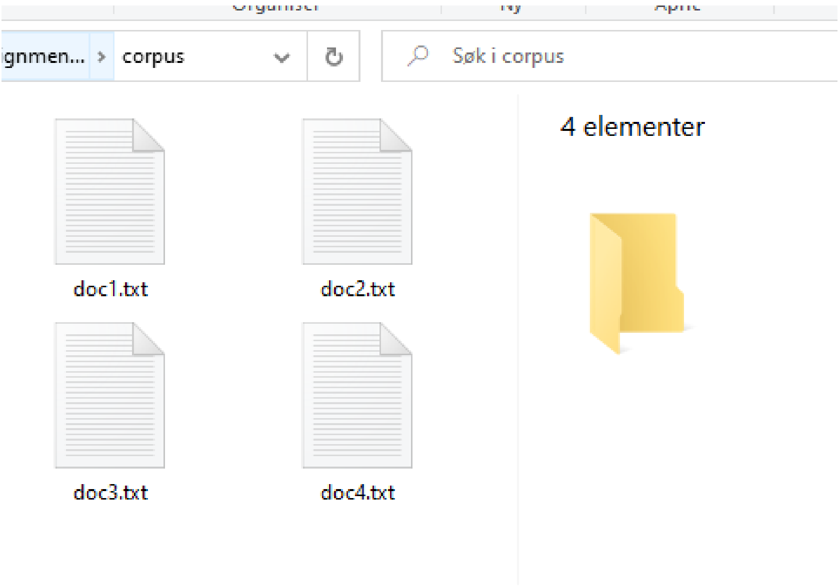
OO-Implementation

- A Document-class
  - Reads the file, and puts each term in a dictionary
    - *Here is a good place to do preprocessing, like remove punctuations etc.
  - Count how many times each word appears in the text
  - Functions for checking if a word is present in the document, sorting, and retrieving the most important words according to TF-IDF

```python
class Document:
    def __init__(self, name, file_path):
        self.name = name
        self.all_terms = {}
        self.word_count = 0

        with open(file_path) as file:
            content = file.read()
            terms = content.split()
            self.word_count = len(terms)
            for t in terms:
                term = t.lower()
                if term in self.all_terms:
                    self.all_terms[term].increase_count()
                else:
                    self.all_terms[term] = Term(term)

    def contains_term(self, term):
        return term in self.all_terms

    def sort_terms(self):
        templist = sorted(self.all_terms.items(), key=lambda x:x[1], reverse = True)
        self.all_terms = dict(templist)

    def get_top_words(self, n):
        top_words = []
        for t in list(self.all_terms)[0:n]:
            top_words.append(t)
        return top_words
```

OO-Implementation

1. Read the files using the Document-class
    1. N = the total number of documents
2. Loop over all documents
    1. Loop over all terms
        1. Count how many of the documents the current term appears in
    2. Compute the TF-IDF of every term
3. Sort the terms of each document based on its TF-IDF value
4. Print out the “most important” words in each document




```python
docs = []
for i in range(1, 5):
    docs.append(Document("doc" + str(i), "\\corpus\\doc" + str(i) + ".txt"))

N = len(docs)

for doc in docs:
    for term in doc.all_terms.items():
        num_occ = 0
        for doc1 in docs:
            if doc1.contains_term(term[1].get_word()):
                num_occ += 1
        term[1].compute_tfidf(doc.word_count, N, num_occ)


for i in range(len(docs)):
    docs[i].sort_terms()
    print("top words in doc" + str(i) + ": ", docs[i].get_top_words(3))
```

Output

C:\Users\Andreas\miniconda3\python.exe

```
top words in doc0:  ['banana', 'yellow', 'new']
top words in doc1:  ['lime', 'green', 'new']
top words in doc2:  ['python', 'new', 'c++']
top words in doc3:  ['document', 'some', 'stuff']
Press any key to continue . . .
```

Further reading

- The String-to-String Correction Problem (Levenshtein) https://dl.acm.org/doi/pdf/10.1145/321796.321811