University of Groningen

Introduction to Data Science

Assignment 7: Text Analysis

Group 16:
Otte Tjepkema *(s3237184)*
José Rodrigues *(s4169328)*
Andrei Miculita *(s4161947)*
Robert Riesebos *(s3220672)*

October 21, 2019

Contents

1 Part A - LSA Search

(a) To create the query vector, i.e. the vector containing the term frequency for each term in the query, we loop over the list of tokens `q` and for each token we call `search_for_term()` on the `InvertedIndex` object. If it returns a tuple that's not `None` we increment the frequency of the term in the query vector, at the corresponding index. Passing the query `"Human Computer Interaction"` results in the query vector: [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0].

```
def create_query_vector(self, q):
    ret_q = [0.0 for i in range(self.I.get_total_terms())]

    # This is where we ensure the query text is tokenized and
    ↪  processed the same as the text was in the Inverted
    ↪  Index (we use the same function call).  It's important
    ↪  any terms we introduce as a query are in the same
    ↪  format as what can be found in the semantic space (if
    ↪  they are there at all).
    q = self.I.process_text(q)

    # Create a query vector
    for t in q:
        w, index_of_t = self.I.search_for_term(t)
        if index_of_t is not None:
            ret_q[index_of_t] += 1

    ##
    # If our semantic space was generated from an initial
    ↪  weighted A matrix1 than we would need to weight our
    ↪  query vector the same way.  You'd do that here.
    ##

    ret_q = self.fold_in_query(ret_q)

    return ret_q
```

Once the query vector is constructed we call `fold_in_query()` with the query vector as argument, in order to transform it into the semantic space (by *folding it in* to its linear space). This is done using the formula given in the assignment: transformed query vector = $\vec{q}^T T S^{-1}$. In code this translates to looping over the number of terms and amount of ranks, and multiplying the passed query vector `q` with `T` and `S_inv`. This method transforms the query vector listed above in to the following vector: [-0.138, 0.0276, -0.0529, 0.614, -0.142, -0.456, 0.261, 0.00335, -0.236] (numbers rounded to three significant digits to prevent cluttering the report).

```python
def fold_in_query(self, q):
    # new query is qTS^{-1}
    no_terms = self.T.shape[0]
    rank = self.T.shape[1]

    # Create vector for folded in query
    fol_q = [0.0 for i in range(rank)]

    # The T matrix will have as many rows as corresponds to
    ↪  terms but will have *rank* number of columns.  Perform
    ↪  the calculation as outlined above.

    # Can you optimise this further using the S_inv array in
    ↪  the constructor?

    # Multiply q by T and scale by S^{-1}
    # qT = q @ self.T
    # fol_q = qT @ self.S_inv
    for i in range(rank):
        for j in range(no_terms):
            fol_q[i] += q[j] * self.T[j][i] * self.S_inv[i]

    return fol_q
```

(b) To compare the similarity of the query that's transformed into the SVD vector space with a document we complete the `cosine_with_doc` method which returns the cosine similarity score between the two vectors. To do so we first reduce the dimensionality of the query and document vectors by only taking the first `max_dimension` values and rows respectively. Then both vectors are scaled using the `S` vector from the SVD. Next the magnitudes of both scaled vectors is calculated using numpy's `linalg.norm()` function. Finally, in order to calculate the cosine similarity score, we use a loop to calculate the dot product of `query_scaled` and `document_scaled` divided by the product of magnitudes for normalization. Which equates to the formula given in the assignment:

$$cos(\theta) = \frac{\vec{q} \cdot D_i^\top}{||\vec{q}||\ \ ||D_i^\top||}$$

```python
def cosine_with_doc(self, q, doc):
```

```
# Dimensionality reduction
query = q[:self.max_dimension]
document = self.Dt[:self.max_dimension, doc]

# Scaling
query_scaled = query * self.S[:self.max_dimension]
document_scaled = document * self.S[:self.max_dimension]

# m_q and m_d are the magnitudes of the query and document
↪  vectors
m_q = np.linalg.norm(query_scaled)
m_d = np.linalg.norm(document_scaled)

# This would be the dot product of q*S [dot] Dt[*]*S[doc]
calc = 0.0

# Can you optimise this further using the S_sq array
↪  generated in the constructor?

# Calculate cosine similarity score
for i in range(self.max_dimension):
    calc += (query_scaled[i] * document_scaled[i]) / (m_q *
    ↪  m_d)
return calc
```

The results of comparing the transformed query vector with all documents are shown in table 1 below.

Document	Cosine similarity score
C3	0.998
C4	0.987
C5	0.908
M1	-0.124
M2	-0.106
M3	-0.099
M4	0.050

Table 1: Similarities between query and documents in semantic space

(c) Finally we implement the method `cosine_with_term` that does the same as the previous `cosine_with_doc` method but it compares two terms instead of a query in semantic space and a document. Hence the code is the same apart from where the query and document are selected and reduced in dimensionality. Here, instead both terms are selected from the `T` vector with the dimensionality corresponding to `max_dimension`.

```
def cosine_with_term(self, t1, t2):
    # Dimensionality reduction
    term_1 = self.T[t1, :self.max_dimension]
    term_2 = self.T[t2, :self.max_dimension]

    # Scaling
    term_1_scaled = term_1 * self.S[:self.max_dimension]
    term_2_scaled = term_2 * self.S[:self.max_dimension]

    # m_t1 and m_t2 are the magnitudes of the term vectors
    m_t1 = np.linalg.norm(term_1_scaled)
    m_t2 = np.linalg.norm(term_2_scaled)

    # This would be the dot product of t1*S [dot] t2*S
    calc = 0.0

    # Can you optimise this further using the S_sq array
    ↪  generated in the constructor?

    # Calculate cosine similarity score
    for i in range(self.max_dimension):
        calc += (term_1_scaled[i] * term_2_scaled[i]) / (m_t1 *
        ↪  m_t2)
    return calc
```

The results of comparing all terms with each other are given in the tables below.

term: *computer*	cos
computer	1.000
eps	0.888
graph	0.210
human	0.874
interface	0.919
minors	0.226
response	0.987
survey	0.793
system	0.946
time	0.987
trees	0.169
user	1.000

term: *eps*	cos
computer	0.888
eps	1.000
graph	-0.264
human	1.000
interface	0.997
minors	-0.248
response	0.801
survey	0.423
system	0.989
time	0.801
trees	-0.304
user	0.900

term: *graph*	cos
computer	0.210
eps	-0.264
graph	1.000
human	-0.291
interface	-0.193
minors	1.000
response	0.366
survey	0.762
system	-0.119
time	0.366
trees	0.999
user	0.182

term: *human*	cos
computer	0.874
eps	1.000
graph	-0.291
human	1.000
interface	0.995
minors	-0.275
response	0.784
survey	0.398
system	0.985
time	0.784
trees	-0.330
user	0.888

term: *interface*	cos
computer	0.919
eps	0.997
graph	-0.193
human	0.995
interface	1.000
minors	-0.177
response	0.842
survey	0.488
system	0.997
time	0.842
trees	-0.234
user	0.929

term: *minors*	cos
computer	0.226
eps	-0.248
graph	1.000
human	-0.275
interface	-0.177
minors	1.000
response	0.381
survey	0.773
system	-0.102
time	0.381
trees	0.998
user	0.198

term: *response*	cos
computer	0.987
eps	0.801
graph	0.366
human	0.784
interface	0.842
minors	0.381
response	1.000
survey	0.881
system	0.881
time	1.000
trees	0.326
user	0.982

term: *survey*	cos
computer	0.793
eps	0.423
graph	0.762
human	0.398
interface	0.488
minors	0.773
response	0.881
survey	1.000
system	0.552
time	0.881
trees	0.735
user	0.775

term: *system*	cos
computer	0.946
eps	0.989
graph	-0.119
human	0.985
interface	0.997
minors	-0.102
response	0.881
survey	0.552
system	1.000
time	0.881
trees	-0.160
user	0.955

term: *time*	cos
computer	0.987
eps	0.801
graph	0.366
human	0.784
interface	0.842
minors	0.381
response	1.000
survey	0.881
system	0.881
time	1.000
trees	0.326
user	0.982

term: *trees*	cos
computer	0.169
eps	-0.304
graph	0.999
human	-0.330
interface	-0.234
minors	0.998
response	0.326
survey	0.735
system	-0.160
time	0.326
trees	1.000
user	0.141

term: *user*	cos
computer	1.000
eps	0.900
graph	0.182
human	0.888
interface	0.929
minors	0.198
response	0.982
survey	0.775
system	0.955
time	0.982
trees	0.141
user	1.000

2 Part B - Term Weightings

(a) **TF-IDF** We start with the matrix A, where A_{ij} corresponds to the number of times term i appears in document j. This matrix uses a term frequency model (TF).

$$A = \begin{bmatrix} 1.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 1.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 1.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 0.00 & 1.00 & 1.00 & 2.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 1.00 & 1.00 & 0.00 \\ 0.00 & 1.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix}$$

Next we will apply two weighing schemes to this matrix. We start with Term Frequency - Inverse Document Frequency (TF-IDF), which is given by

$$A_{ij} = f_{ij} \log \left(\frac{n}{\sum_j \chi(f_{ij})} \right)$$

where

$$\begin{aligned} f_{ij} &= \text{The term frequency of term } i \text{ in document } j \\ \chi(\nu) &= 1 \text{ if } \nu > 0, 0 \text{ otherwise (binary indicator)} \\ n &= \text{Total number of documents} \end{aligned}$$

note also that $\sum_j$ indicates the sum over one row of the matrix. To apply this formula a simple matlab script was made, which can be seen in section 2.1. Since we are dealing with matrices a nested for loop was used. To calculate the $\sum_j \chi(f_{ij})$ term, the number of nonzero entries in row i was taken. The result of this loop gives us a matrix containing the TF-IDF terms, which we call matrix B.

$$B = \begin{bmatrix} 1.50 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.50 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.10 & 1.10 & 1.10 \\ 1.50 & 0.00 & 0.00 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 1.50 & 0.00 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.50 & 1.50 \\ 0.00 & 1.50 & 0.00 & 0.00 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.50 \\ 0.00 & 1.10 & 1.10 & 2.20 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.50 & 0.00 & 0.00 & 1.50 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.10 & 1.10 & 1.10 & 0.00 \\ 0.00 & 1.10 & 1.10 & 0.00 & 1.10 & 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix}$$

(b) **Log-Entropy** Finally we apply the log entropy weighing scheme, which is given by

$$\mathrm{A}_{ij} = \log(1 + f_{ij}) \left[1 + \left(\sum_j \frac{p_{ij} \log(p_{ij})}{\log(n)} \right) \right]$$

where

$$\begin{aligned} f_{ij} &= \text{The term frequency of term } i \text{ in document } j \\ p_{ij} &= \frac{f_{ij}}{\sum_j f_{ij}} \\ &\quad \text{Proportion of term } i \text{ in document } j \text{ with respect} \\ &\quad \text{to the total frequency count of } f_{ij} \\ n &= \text{Total number of documents} \end{aligned}$$

where again $\sum_j$ implies a sum over the columns of matrix A. The Log-Entropy matrix was also calculated with the matlab script. To simplify the calculation another matrix containing the p_{ij} terms was first computed. After that we can easily compute the Log-Entropy matrix by using a nested for loop. Note that wen $\mathrm{A}_{ij} = 0$, $p_{ij} \log p_{ij} = 0 * -\infty$, which is not defined, to solve this we use the limit definition

$$\lim_{x \to 0} x \log(x) = 0$$

which can be found using l'Hospital's rule. Therefore we replace the $0 * -\infty$ elements with 0. This gives us the matrix containing Log-entropy elements, which we call matrix C

$$C = \begin{bmatrix} 0.47 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.47 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.35 & 0.35 & 0.35 \\ 0.47 & 0.00 & 0.00 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.47 & 0.00 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.47 & 0.47 \\ 0.00 & 0.47 & 0.00 & 0.00 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.47 \\ 0.00 & 0.37 & 0.37 & 0.58 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.47 & 0.00 & 0.00 & 0.47 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.35 & 0.35 & 0.35 & 0.00 \\ 0.00 & 0.35 & 0.35 & 0.00 & 0.35 & 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix}$$

2.1 Matlab code

```
%Construct matrix A
A = [1,1,0,0,0,0,0,0,0;
     0,0,1,1,0,0,0,0,0;
     0,0,0,0,0,0,1,1,1;
     1,0,0,1,0,0,0,0,0;
     1,0,1,0,0,0,0,0,0;
     0,0,0,0,0,0,0,1,1;
     0,1,0,0,1,0,0,0,0;
     0,1,0,0,0,0,0,0,1;
     0,1,1,2,0,0,0,0,0;
     0,1,0,0,1,0,0,0,0;
     0,0,0,0,0,1,1,1,0;
     0,1,1,0,1,0,0,0,0];
%Construct matrix B for TF-IDF
[nrow,ncol] = size(A);
B = zeros(nrow,ncol);
for i = 1:nrow
    sumChi = sum(any(A(i,:),1));
    for j = 1:ncol
        B(i,j) = A(i,j)*log(ncol/sumChi);
    end
end

%Construct matrix C for Log-Entropy
```

```
C = zeros(nrow,ncol);
P = zeros(nrow,ncol); %Create p_ij elements in matrix P
for i = 1:nrow
    sumf = sum(A(i,:));
    for j = 1:ncol
        P(i,j) = A(i,j)/sumf;
    end
end

for i = 1:nrow
    pij = P(i,:);
    log_pij = log(P(i,:));
    log_pij(log_pij==-Inf) = 0;
    sumterm = sum(pij.*log_pij);
    for j = 1:ncol
        C(i,j) = log(1+A(i,j))*(1+sumterm/log(ncol));
    end
end
```