Topic Modeling

Neba Nfonsang
University of Denver

Objectives

- Find topics in documents
- Perform explicit semantics analysis
- Document clustering
- Latent semantic analysis
- Non-negative matrix factorization

Topic Modeling

- NLP is commonly used to search and extract information.
- Topic modeling involves automatically defining the subject of a document based on the important words in the document.
- We have already seen how to represent the relationship between words and documents using TF-IDF weights.
- A matrix (a collection of vectors) can be used to capture the statistical relationship between words and documents.

Term-Document Matrix

- A term-document matrix represents the relationship between words and documents.
- The rows are words while the columns are documents.
- A text or corpus of documents can be represented using a matrix.
- When we use a matrix to represent a corpus of documents, each column is a document and each row corresponds to a unique word in the text.

Term-Document Matrix (TDM)

	d1	d2	d3
w1	1	0	1
w2	1	1	0
w3	0	1	0
w4	0	0	0
w5	1	1	0
w6	1	1	1

- The values in the term document matrix could be Boolean values indicating whether the word is present in the document or not
- The values in the TDM could be TF-IDF weights which indicate relative importance of the word in the document.
- Frequency counts could be used as well.

Explicit Semantic Analysis (ESA)

	d1	d2	d3
w1	w1d1	0	w1d3
w2	w2d2	w2d2	0
w3	0	w3d2	0
w4	0	0	0
w5	w5d2	w5d2	0
w6	w6d2	w6d2	w6d2

- ESA involves vectoral representation of text (words or documents) where the vector entries are typically tf-idf weightings.
- So, ESA starts with creating a Term Document Matrix of tf-idf weights. **widj** are tf-idf values of word i on document j and zero values mean the word was absent in that document.

ESA: Word and Document Similarity

- Explicit Semantic Analysis is an approach used to find similarity (relatedness) between words and documents
- In ESA, TDM is used and typically, the entries of the vectors are the tf-idf weights (relative importance) of words on documents.
- The documents or words in the TDM can be expressed as vectors and the similarity between words or documents can be computed using:
- $\text{sim}(\text{vec1}, \text{vec2}) = \frac{\text{vec1} \bullet \text{vec2}}{|\text{vec1}| * |\text{vec2}|}$

ESA: Documents as Vectors

	d1	d2	d3
w1	1	0	1
w2	1	1	0
w3	0	1	0
w4	0	0	0
w5	1	1	0
w6	1	1	1

- The documents can be represented as column vectors:
  - d1 = (1, 1, 0, 0, 1, 1)
  - d2 = (0, 1, 1, 0, 1, 1)
  - d3 = (1, 0, 0, 0, 0, 1)
- Ones and zeros are used for simplicity. Practically, the actual tf-idf values should replace the ones.

ESA: Documents as Vectors

	d1	d2	d3
w1	1	0	1
w2	1	1	0
w3	0	1	0
w4	0	0	0
w5	1	1	0
w6	1	1	1

- **Document similarity**: as already noted, document similarity can be analyzed using cosine similarity formula.
- Larger cosine values (lesser angles) indicates more similarity, which implies most words in one document appear in the other document, hence similarity in context. sim(vec1, vec2) = cos 90 = 0)

ESA and Document Similarity

- d1 = (1, 1, 0, 0, 1, 1)
- d2 = (0, 1, 1, 0, 1, 1)
- d3 = (1, 0, 0, 0, 0, 1)
- Given that the documents are represented as vectors, explicit semantic analysis finds the relationship between the document vectors.
- The idea is that, documents with similar corresponding vector entries are similar. So the cosine similarity is used to find the angle between the vector. Similar vectors have a smaller angle or “distance” between them.

ESA and Document Similarity

- d1 = (1, 1, 0, 0, 1, 1)
- d2 = (0, 1, 1, 0, 1, 1)
- d3 = (1, 0, 0, 0, 0, 1)
- Given the vectors representing documents in a corpus, the similarity between documents can be found using **cosine similarity.**

- The similarity between two vectors or documents could be found using the formula:
- $\text{sim}(\text{vec1}, \text{vec2}) = \frac{\text{vec1} \bullet \text{vec2}}{|\text{vec1}|\,|\text{vec2}|}$
- Note that 1-cosine similarity is called **cosine distance.**

ESA: Words as Vectors

	d1	d2	d3
w1	1	0	1
w2	1	1	0
w3	0	1	0
w4	0	0	0
w5	1	1	0
w6	1	1	1

- A word vector is a row vector with entries corresponding to the tf-idf weights of words across the documents.
- We can draw conclusions on the **relatedness** of two words based on their cosine similarity value, which captures similarity in the meaning of the two words

ESA: Word Vectors and Similarity

	d1	d2	d3
w1	1	0	1
w2	1	1	0
w3	0	1	0
w4	0	0	0
w5	1	1	0
w6	1	1	1

- **Word similarity:** note that the meaning of a word can be inferred through the document or concept it contributes to.
- Terms or words with close meaning tend to be located together (appear across the same documents)

Using ESA for Document Classification

- So, ESA can be used for text or document classification. Mostly, content-based classification.
- If we have documents that are labelled or indexed in the database, we can compare a new document to the database documents, then classify the new document.
- Sometimes the documents are classified according to their attributes including genre, document type, author, year, etc.

Using ESA for Information Retrieval

- ESA is used for information retrieval.
- The Term Document Matrix of the corpus is used as the knowledge base.
- A new document is then used to query the knowledge base to find related documents.
- Basically, the new document is compared to all the documents in the knowledge base using the cosine similarity.
- Cosine similarity values range from 0 to 1 given that the vectors had positive entries.

Using ESA for Information Retrieval

- When a new document has been tokenized and is now represented as a list of words, we can convert this list of words into a vector by summing or averaging all the word vectors in the database for each of the words in the new document.
- The entire English Wikipedia is the knowledge base corpus , so the assumption here is that the words in the new documents are also in the database or TDM matrix.

Using ESA for Information Retrieval

- If new document was
- new_doc = (w1, w2, w3)
- Then the word vector for this new document will be: new_vector = w1 + w2 + w3
- This can be done in code by initializing the new vector with zero entries and dimensions = length of a row. Augmented statements can be used to add the vectors from the TDM

	d1	d2	d3
w1	1	0	1
w2	1	1	0
w3	0	1	0
w4	0	0	0
w5	1	1	0
w6	1	1	1

Using ESA for Information Retrieval

- Since the dimensions of the word vector correspond to documents, we can use the dimension with the highest tf-idf score to retrieve the document corresponding to this score as the related document
- On the other hand, we can find similar words by iterating through the rows of TDM and computing the cosine similarity of each row or word vector and the new word vector. The row in the database with highest cosine similarity represent the most similar words

Implementing Term Document Matrix

use the TFIDF matrix and the vocabulary size to get the Term Document Matrix

```python
def term_document_matrix(TFIDF, word_list, word_dict):
    # how many unique words are in the text
    vocabulary_size = len(word_dict)
    number_documents = len(TFIDF)

    # create a numpy array with shape being
    # vocab size for row num and number of doc for cols
    TDM = np.zeros((vocabulary_size, number_documents))

    # select each document in TFIDF
    for doc_num in range(number_documents):
        document = TFIDF[doc_num]

        # for each word in the doc, get the
        # int label from word_dict
        for word in document.keys():
            int_label = word_dict[word]

            # get the value of TFIDF and
            # assign it to the positions
            # determined by int_label and doc_num
            TDM[int_label, doc_num] = document[word]

    return TDM
```

Note that the term document matrix can have words apearing in one document and not in the other. Wherever the word did not appear, there would be a numpy zero value

```python
TDM = term_document_matrix(TFIDF, word_list, word_dict)
print((f"the dataset has {TDM.shape[0]} unique words and " +
       f"{TDM.shape[1]} documents"))
```

```
the dataset has 927 unique words and 200 documents
```

```python
TDM.shape
```

```
(927, 200)
```

```python
# in the term document matrix, the rows are the words
len(word_list)
```

```
927
```

Term Document Matrix

```
1  TDM[:10,:5]
```

```
array([[0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 5.29831737, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
```

Implementing ESA

```python
# define a function that finds related documents
# to the new document

def find_related_docs(tweet, TDM):
    # create a vector representing the new document
    vec_dim = TDM.shape[1] # dimension of vector
    # fill the vectors with zeroes
    # this would be a row vector representing the word
    new_vector = np.zeros(vec_dim)

    for word in tweet:
        # get the integer label or position of word
        # the position matches with the row position of word on TDM
        position = word_dict[word]
        # extract rows with the words in tweets and add them
        # the entries with the highest tf-idf
        # corresponds to the most related document
        new_vector = new_vector + TDM[position, :]

    # label the tf-idfs of the new vector with integers
    # then sort the tf-idf entries of the new vector
    doc_list = sorted(zip(range(vec_dim), new_vector),
                      key=lambda x:x[1], reverse=True)
    return doc_list
```

```python
# the most related document has the highest total
# for tf-idf. This is if all or most words in the
# new document appear in a particular document, that document
# will have values of tf-idf leading to higher tf-idf totals

related_docs = find_related_docs(new_tweet, TDM)
related_docs[:10]
```

```
[(17, 6.725433722188182),
 (34, 6.725433722188182),
 (2, 5.298317366548036),
 (166, 3.506557897319982),
 (167, 3.506557897319982),
 (190, 3.506557897319982),
 (191, 3.506557897319982),
 (154, 3.2188758248682006),
 (164, 3.2188758248682006),
 (165, 3.2188758248682006)]
```

Documents Related to Search Query

```
# print the top 5 documents related to the
# new document used for search query

print("Relatiated Documents:")

for int_label, total_tfidf in related_docs[0:5]:
    print("-----------------------------------------------------------------------------------")
    print(int_label, tweets[int_label])
```

```
Relatiated Documents:
-----------------------------------------------------------------------------------
17 ['jh', 'hines', 'staff', 'newly', 'issued', '@apple', '#connected', 'macbook', 'ipad', 'mini', '#txed']
-----------------------------------------------------------------------------------
34 ['rt', '@tra_hall', 'jh', 'hines', 'staff', 'newly', 'issued', '@apple', '#connected', 'macbook', 'ipad', 'mini', '#txed',
'#txed']
-----------------------------------------------------------------------------------
2 ['#aapl:5', 'rocket', 'stocks', 'buy', 'december', 'gains', 'apple']
-----------------------------------------------------------------------------------
166 ['ipad', 'mini', 'unboxing', 'via', '@youtube', '@apple', '#ipadmini', '#ipad', '#macbook', '#macbookpro', '#startup', '#h
ipster', '#unboxing']
-----------------------------------------------------------------------------------
167 ['ipad', 'mini', 'first', 'time', 'startup', 'via', '@youtube', '@apple', '#ipadmini', '#ipad', '#macbook', '#macbookpro',
'#startup', '#hipster', '#unbox']
```

Alternatively, Find Related Words to New Vector: Cosine Similarities

```python
# Alternatively, use similarity cosine to find related documents

def similarity(vec1, vec2):
    """
    compute cosine similarity between vectors
    """
    sim = np.dot(vec1, vec2)
    norm1 = np.sqrt(np.dot(vec1, vec1))
    norm2 = np.sqrt(np.dot(vec2, vec2))
    cosine_sim = sim/(norm1*norm2)

    return cosine_sim

def find_similar_words(tweet, TDM):
    vec_dim = TDM.shape[1] # dimension of vector
    # initialize new vector for new tweet
    new_vector = np.zeros(vec_dim)

    # create new vector (word vector) for new tweet
    for word in tweet:
        position = word_dict[word]
        new_vector = new_vector + TDM[position, :]

    # similarity between new vector and each word vector
    sim_words = [similarity(new_vector, TDM[i, :]) for i in range(TDM.shape[0])]
    sim_words_sorted = sorted(zip(range(TDM.shape[0]), sim_words),
                    key=lambda x:x[1], reverse=True)

    return sim_words_sorted

```

```python
similar_words = find_similar_words(new_tweet, TDM)
similar_words[:10]
```

```
[(593, 0.7640071602554539),
 (346, 0.7398422630773179),
 (518, 0.6849608378181927),
 (863, 0.66219576837537),
 (526, 0.6273056189756162),
 (581, 0.6273056189756162),
 (844, 0.6273056189756162),
 (381, 0.6273056189756161),
 (460, 0.6273056189756161),
 (537, 0.6273056189756161)]
```

Words that are related occur together across documents

Related Words

```
similar_words = find_similar_words(new_tweet, TDM)
similar_words[:10]
```

```
[(593, 0.7640071602554539),
 (346, 0.7398422630773179),
 (518, 0.6849608378181927),
 (863, 0.66219576837537),
 (526, 0.6273056189756162),
 (581, 0.6273056189756162),
 (844, 0.6273056189756162),
 (381, 0.6273056189756161),
 (460, 0.6273056189756161),
 (537, 0.6273056189756161)]
```

```
# print similar words
for int_label, sim_score in similar_words[0:10]:
    print(word_list[int_label], sim_score)

```

```
macbook 0.7640071602554539
mini 0.7398422630773179
ipad 0.6849608378181927
#macbook 0.66219576837537
#startup 0.6273056189756162
@youtube 0.6273056189756162
#hipster 0.6273056189756162
newly 0.6273056189756161
jh 0.6273056189756161
issued 0.6273056189756161
```

Document Clustering

- We can group documents based on their content. If the documents have similar contents, we can group them together.
- Before we group documents we need to specify how we want to measure the distance between them. – use cosine similarity

- Using the cosine similarity:

$$\text{sim}\left(\vec{u}, \vec{v}\right) = \frac{\vec{u} \cdot \vec{v}}{||\vec{u}|| \, ||\vec{v}||}$$

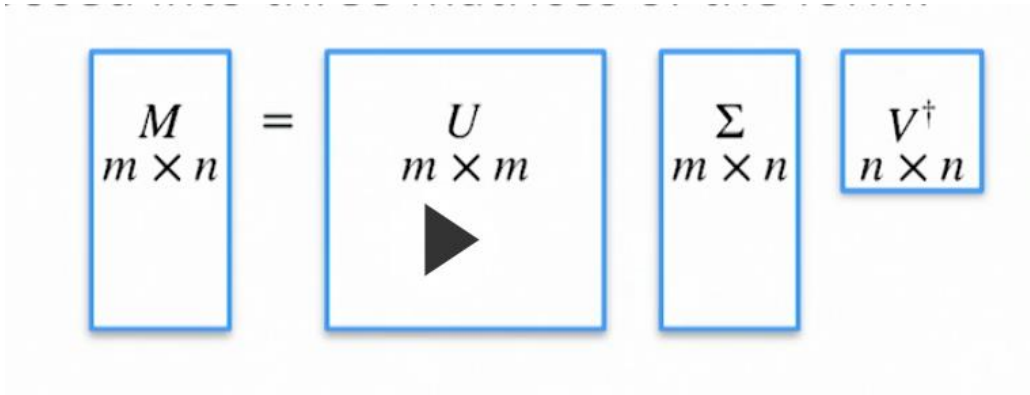
a simple approach is to impose a minimum similarity cut off and any value above the cut off implies the documents in question are similar

Document Clustering

- This is how the algorithm works:
- Every document is assigned to its own cluster
- The matrix similarity are calculated.
- The two most similar vectors are merged.
- Similarity is recalculated for the reduced set of clusters until a desired number of clusters is reached.

Latent Semantic Analysis (LSA)

- LSA, sometimes called latent semantic indexing, is a data reduction technique that uses singular value decomposition to extract latent dimensions from the data, which are compact representation of the



where
- U is a unitary matrix ($UU^\dagger = 1$)
- Σ is diagonal with non-negative elements
- $V^\dagger$ is a unitary matrix

Symmetric S	$S = Q\Lambda Q^{\mathrm{T}} = \lambda_1 \boldsymbol{q}_1 \boldsymbol{q}_1^{\mathrm{T}} + \lambda_2 \boldsymbol{q}_2 \boldsymbol{q}_2^{\mathrm{T}} + \cdots + \lambda_r \boldsymbol{q}_r \boldsymbol{q}_r^{\mathrm{T}}$
Any matrix A	$A = U\Sigma V^{\mathrm{T}} = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^{\mathrm{T}} + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^{\mathrm{T}} + \cdots + \sigma_r \boldsymbol{u}_r \boldsymbol{v}_r^{\mathrm{T}}$

LSA and Latent Dimensions

- With singular value decomposition, the term-document matrix is decomposed into components and the first few components (k dimensions) with the maximum variance are extracted such that the variance in the data is preserved.
- The k dimensions selected represent k latent topics or dimensions of the underlying dataset or TDM
- Only the most significant dimensions are kept to get rid of the noise. We want to represent the most information in the dimensions we select.

LSA and SVD

- In LSA, singular value decomposition (SDV) is applied to the term-document matrix to obtain a SVD term matrix U, the diagonal matrix Σ and SDV document matrix V.

Example of Term Document Matrix

		d_1	d_2	d_3	d_4	d_5	d_6	
	ship	1	0	1	0	0	0	
	boat	0	1	0	0	0	0	
	ocean	1	1	0	0	0	0	
	voyage	1	0	0	1	1	0	
	trip	0	0	0	1	0	1	

https://nlp.stanford.edu/IR-book/html/htmledition/latent-semantic-indexing-1.html

LSA and SVD

SVD term matrix (U)

	1	2	3	4	5
ship	−0.44	−0.30	0.57	0.58	0.25
boat	−0.13	−0.33	−0.59	0.00	0.73
ocean	−0.48	−0.51	−0.37	0.00	−0.61
voyage	−0.70	0.35	0.15	−0.58	0.16
trip	−0.26	0.65	−0.41	0.58	−0.09

Diagonal matrix

$$\begin{bmatrix} 2.16 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.59 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.28 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.39 \end{bmatrix}$$

SVD document matrix

	d_1	d_2	d_3	d_4	d_5	d_6
1	−0.75	−0.28	−0.20	−0.45	−0.33	−0.12
2	−0.29	−0.53	−0.19	0.63	0.22	0.41
3	0.28	−0.75	0.45	−0.20	0.12	−0.33
4	0.00	0.00	0.58	0.00	−0.58	0.58
5	−0.53	0.29	0.63	0.19	0.41	−0.22

Now, we can multiply the matrices and consider only a few, say two singular values. That means we are selecting only the first two components. It is like “zeroing” out the rest of the singular values in the diagonal matrix after the second one, then multiply the matrices. So the term document matrix is approximated using k-dimensional singular values

LSA: projecting a new document to a singular space

- Documents and words can be easily classified in the singular space.
- A new document (or query) is mapped or projected into the singular space using the transformation:

$$\hat{v} = \Sigma_k^{-1} U_k^{\dagger} v$$

- v is the original document
- v-bar is the transformed version of the original document
- $U_k \; and \; \Sigma_k$ are the truncated versions of U and Σ. When we select k=2 components, all other rows after the 2 rows are zeroed out to get the truncated versions.

Examples of Truncated Σ_k

2.16	0.00	0.00	0.00	0.00
0.00	1.59	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
0.00	0.00	0.00	0.00	0.00

- If k=2, only the first two rows of the matrices are maintained. The truncated version therefore has the rest of the rows filled with zeros.

Implementing LSA

- Most packages such as sk-learn have the svd feature for decomposing a matrix into its three components: In LSA, SDV needs to be first applied to the TDM:

```
u, sigma, vt = svd(TDM)
```

Shapes of Matrices

- m, n = TDM.shape
- print(sigma.shape)
- print(vt.shape)
- Note that: $TDM = U\Sigma V^T$
- sdv can be imported as follows:
- from scipy.linalg import svd

Implementing SVD

- When we decompose the term document matrix, we get

```
U (m x m) . Sigma (n x n) . V^T (n x n)
```

- But we actually require:

```
U (m x m) . Sigma (m x n) . V^T (n x n)
```

- So we can reconstruct the diagonal matrix into an mxn matrix
- We could decide to keep sigma as nxn and reconstruct u to mxn. So we can have mxn, nxn, nxn

LSA Implementation in Python

Latent Semantic Analysis

This is the extraction of latent dimensions or topics from a singular decomposition of the term-document matrix

```
1  # starts with the term-document matrix
2  TDM
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

Implement SVD

```
# decompose the term-document matrix
u, sigma, vt = svd(TDM)
```

```
# print the svd term matrix
u
```

```
array([[ 7.36806317e-05,  3.40770148e-04, -7.77823847e-03, ...,
         1.05289194e-02, -7.46998095e-05,  6.69916034e-04],
       [ 3.44239397e-05,  3.98821846e-05, -2.23763614e-05, ...,
         3.75624127e-04,  1.25668498e-04,  8.11037736e-05],
       [ 7.07206337e-03,  8.57493541e-04, -1.24079164e-02, ...,
         1.04724698e-03,  5.16283990e-04,  6.36249565e-04],
       ...,
       [ 3.18213574e-04,  1.54391576e-03, -1.40497485e-02, ...,
         8.24330670e-01, -6.80895387e-05,  1.20092051e-04],
       [ 5.63686605e-05,  4.49917970e-04, -1.57398765e-03, ...,
        -2.17853697e-03,  8.78420001e-01, -1.53654818e-04],
       [ 4.61032665e-05,  6.28857248e-05, -4.27434654e-05, ...,
         1.25491796e-04,  9.38039129e-03,  9.17042436e-01]])
```

```
# print sigma
sigma[:5]
```

```
array([34.12143118, 28.69781728, 25.40239066, 24.53357897, 22.68779674])
```

```
# print the svd document matrix
vt
```

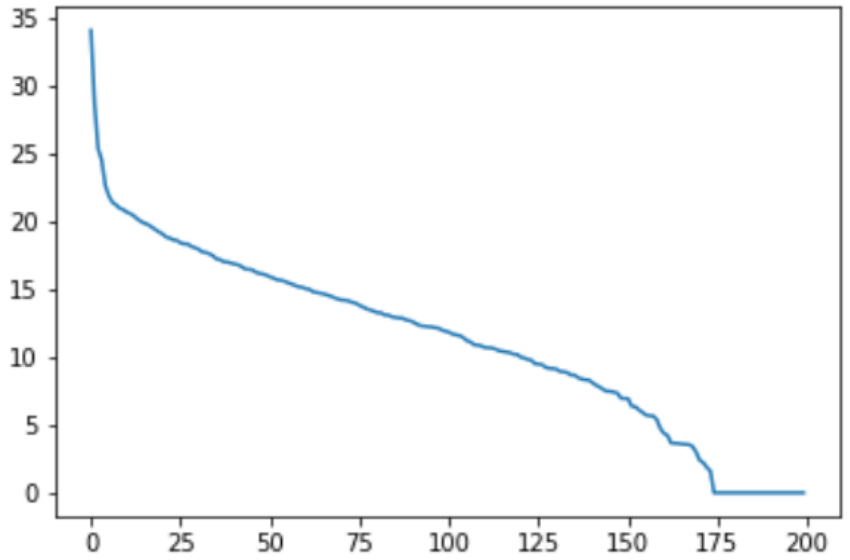
```
array([[ 2.33523602e-04,  2.10340123e-04,  1.88044005e-04, ...,
         1.57925095e-03,  2.19588538e-03,  7.71782278e-04],
       [ 2.33069252e-04,  1.31105677e-03,  2.16875708e-03, ...,
         1.70972476e-02,  1.20388328e-01,  5.43102303e-03],
       [-1.18582648e-04, -2.57411317e-02, -2.27348622e-02, ...,
        -1.45939720e-03, -3.54534831e-02, -1.02066596e-01],
       ...,
       [ 0.00000000e+00, -9.73800447e-17, -1.32165671e-16, ...,
         1.14491749e-16,  5.20417043e-17,  1.73472348e-17],
       [ 0.00000000e+00,  1.86049093e-16,  2.06215253e-16, ...,
         5.81132364e-17, -4.51028104e-17, -3.46944695e-18],
       [ 0.00000000e+00,  8.84708973e-17,  6.93889390e-17, ...,
        -9.71445147e-17, -5.37764278e-17, -1.38777878e-17]])
```

```
# print the shapes of the matrices
print(TDM.shape, u.shape, sigma.shape, vt.shape)
```

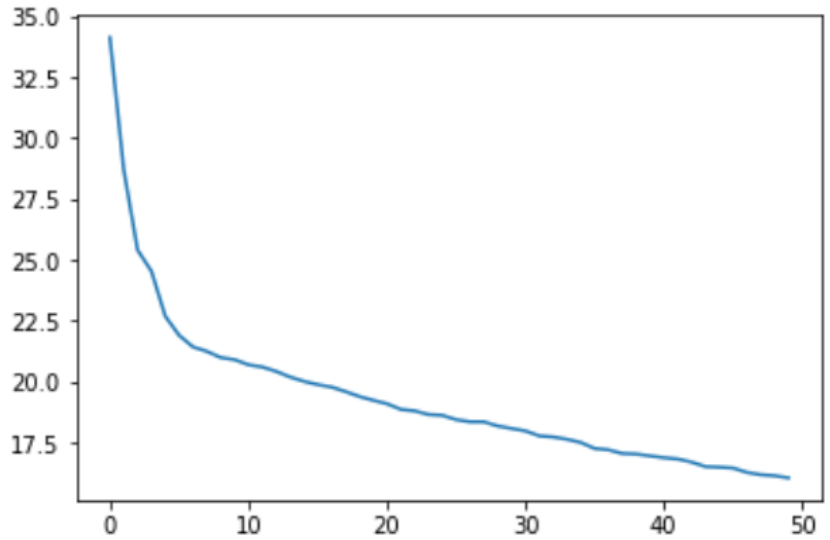
```
(927, 200) (927, 927) (200,) (200, 200)
```

How Many Dimensions Should be Extracted?

```
# it is difficult to see the precise "elbow" value
# when there are several values of sigma
plt.plot(sigma)
plt.show()
```




```
plt.plot(sigma[:50])
plt.show()
```



The graph shows that k=5 would be appropriate, from the "elbow" value

Creating Truncated Matrices with K Dimensions

```
# from the shape of sigma, it can be noticed that sigma is a vector,
# sigma is supposed to be a matrix mxn, we can reconstruct it if we wanted
# Let's go straight to converting sigma into a truncated diagonal matrix

k = 5
sigma_k = sigma[:k]
sk = np.diag(sigma_k)
sk
# this is nxn matrix
```

```
array([[34.12143118,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        , 28.69781728,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , 25.40239066,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , 24.53357897,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 22.68779674]])
```

```
# we need a 927x5 for uk to keep only k-dimensions
uk = u[:, :k]
```

```
# we need 5x200 for vtk to keep only k-dimensions
vtk = vt[:5, :200]
```

```
print(uk.shape, sk.shape, vtk.shape)
```

```
(927, 5) (5, 5) (5, 200)
```

View a few Unique Words in the Dataset

```
1  # print the word list, just a few words out of 927
2  print(word_list[:20])
```

```
['best-designed', 'well', 'broadway', '#iwantjobsback', 'th
ree', 'increases', 'thankful', 'stocks', 'wilson', 'competi
tion', 'charging', 'im', 'bloomberg', '@baba', 'analyst',
'music', 'iphones', 'shareholders', 'reveal', 'self']
```

```
1  len(word_list)
```

```
927
```

Words that Contribute Most, to a Topic

Find the Meaning of an Extrated Topic

When a topic or a vector (column vector) is obtained from the extracted SVD term matrix, we can define what that topic is by mapping the vector (whose entries are latent tf-idf scores) to the wordlist (unique words in the text in the order in which we have them on the term-document matrix) then sorting the list by the latent scores. We want to identify the words that contribute most to a particular topic

```
# map the entries of the latent scores from the
# topic vector to their corresponding words
def top_words(wordlist, topic_vector):
    doc_list = sorted(zip(wordlist, topic_vector),
                      key=lambda x:x[1], reverse=True)

    return doc_list

```

```
# let's find the words that contribute most to topic 1
# this will help us define topic 1
topwords = top_words(word_list, uk[:,0])
for word, weight in topwords[:10]:
    print(word, ": ", weight)
```

```
iphones :  0.2987441726579015
federal :  0.2987441726579015
cites :  0.2987441726579015
department :  0.2987441726579015
century :  0.2987441726579015
unlock :  0.2987441726579015
law :  0.2987441726579015
18th :  0.2987441726579015
justice :  0.2942978100155947
@thehill :  0.26993335660102
```

Transform a Given Document into the Singular Space

```
# given a sentence or a word document,
# let's define the vector in a regular word space
document = ["ipad", "iphone", "mini"]
doc_vector = np.zeros(len(word_list))

# view the first 10 entries
doc_vector[:10]
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
for word in document:
    # extract the index of the word
    integer_index = word_dict[word]
    # update the entry corresponding to
    # the integer index
    doc_vector[integer_index] = 1

    # view a portion of the vector
doc_vector[:200]
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
```

Let's project the vector into a singular or latent space

$$\hat{v} = \Sigma_k^{-1} U_k^{\dagger} v$$

```
# sigma_k^-1 is a singular vector
sigma_k
```

```
array([34.12143118, 28.69781728, 25.40239066, 24.53357897, 22.687
79674])
```

```
1/sigma_k
```

```
array([0.02930709, 0.03484586, 0.03936637, 0.04076046, 0.0440765
6])
```

```
uk.T
```

```
array([[ 7.36806317e-05,  3.44239397e-05,  7.07206337e-03, ...,
         3.18213574e-04,  5.63686605e-05,  4.61032665e-05],
       [ 3.40770148e-04,  3.98821846e-05,  8.57493541e-04, ...,
         1.54391576e-03,  4.49917970e-04,  6.28857248e-05],
       [-7.77823847e-03, -2.23763614e-05, -1.24079164e-02, ...,
        -1.40497485e-02, -1.57398765e-03, -4.27434654e-05],
```

Transform a Given Document into the Singular Space

```
1  # the projected document would be
2  singular_doc = 1/sigma_k*np.dot(uk.T, doc_vector)
3  singular_doc
```

```
array([ 0.00017605,  0.01445142,  0.00050387, -0.0004712 , -0.00121434])
```

Get the Topic Most Related to the Document

Topic most relevant for the document

```
# the entries of the singular vector are on specific dimensions
# in the singular space. We want to find the dimension with the largest
# value, so we use np.argmax which returns the dimension with the largest value
topic = np.argmax(singular_doc)
topic
```

```
1
```

Retrieving Top Words

Retrieve the top words for the topic related to the document

```
1  topwords = top_words(word_list, uk[:,topic])
2  for word, weights in topwords[:10]:
3      print(word, ":",  weight)
```

```
#macbook : 0.26993335660102
#startup : 0.26993335660102
@youtube : 0.26993335660102
#hipster : 0.26993335660102
#macbookpro : 0.26993335660102
via : 0.26993335660102
mini : 0.26993335660102
ipad : 0.26993335660102
#ipadmini : 0.26993335660102
startup : 0.26993335660102
```

ESA and LSA

- ESA uses a corpus, then defines the meaning of a word based on the documents the word contributes the most.
- Original Tf-idf weights are used from the TDM to find words in the corpus related to a query document or find similarity between query doc and docs in corpus
- LSA finds the underlying representation or latent topics in a text. The latent topics can then be labelled based on the words that contribute most to the topics
- A query document can be mapped to the latent space to see which topic is relevant to the document.