Unsupervised techniques for Text Summarization

Summarization based on Term Frequency

1. Input: Text Corpus (unlabeled)

2. Define a **'min sentence length'** and pass it as an input to docSent

3. **docSent** is a **function** which first preprocesses the text, removes non-ASCII terms, tokenizes sentences and then **returns all the sentences whose length is >= 'min sentence length'**

4. Define a **'min summary length'**. If the total len(sentences) obtained from docSent is less than min summary length then just output all the sentences as summary.

5. Next, a term frequency table is created for all the words in the tokenized sentences using TF-IDF

6. For each sentence, the **'sentence score'** is calculated by adding the term frequencies of the words appearing in the sentence

7. Lengths 'L' of all the sentences are calculated. This is done for normalization purposes. If linear normalization to be done then 'sentence score'/L. Log normalization can also be done. In this way, **'normalized sentence scores'** are obtained.

8. Arrange sentences in descending order wrt the 'normalized sentence scores'. The next step is diversification using the **MMR** technique (similarity of sentences based approach). Sentences with higher diversity scores are selected keeping in mind the 'summary length' parameter

9. Output: Sentences in descending order of diversity scores.

Summarization based on Sum Basic

1. Input: Text Corpus (unlabeled)

2. Normalized Sentence scores calculated in the same as the Term Frequency method and sort the sentences in descending order

3. Take sentence with the highest score and append it to the ‘top sentence list’ as it is. Then square the frequencies of all the words appearing in that sentence. (since freq<1 squaring reduces them). Now remove this sentence from the list of available sentences.

4. Point 3 is iterated for i in range(len(summary size)). After each iteration, the frequency of the words is modified by squaring them. In each iteration, normalized scores are calculated of the available sentences and the process is repeated. This is done to promote diversification.

5. After iteration is complete the required summary sentences are obtained in the ‘top sentences list’. Sorting is applied to this list.

6. Output: sentences in descending order of a kind of ‘iteratively modified word frequency/sentence scores’

Summarization based on Latent Semantic Indexing

1. Input: Text Corpus (unlabeled)

2. Define a **‘min sentence length’** and pass it as an input to docSent

3. **docSent** is a **function** which first preprocesses the text, removes non-ASCII terms, tokenizes sentences and then **returns all the sentences whose length is >= ‘min sentence length’**

4. Define a **‘min summary length’**. If the total len(sentences) obtained from docSent is less than min summary length then just output all the sentences as summary.

5. We have the tokenized sentences until this point

6. Create a dictionary of tokenized words using gensim corpora.Dictionary()

7. Create a corpus variable = **bag of words** representation of the tokenized sentences

8. Train the LSI model using. Various parameters that can be tweaked are:
    a. **corpus** (*{iterable of list of (int, float), scipy.sparse.csc}, optional*) – Stream of document vectors or sparse matrix of shape (num_terms, num_documents).
    b. **num_topics** (*int, optional*) – Number of requested factors (latent dimensions)
    c. **id2word** (*dict of {int: str}, optional*) – ID to word mapping, optional.
    d. **chunksize** (*int, optional*) – Number of documents to be used in each training chunk.

e. **decay** (*float, optional*) – Weight of existing observations relatively to new ones.
f. **distributed** (*bool, optional*) – If True - distributed mode (parallel execution on several machines) will be used.
g. **onepass** (*bool, optional*) – Whether the one-pass algorithm should be used for training. Pass False to force a multi-pass stochastic algorithm.
h. **power_iters** (*int, optional*) – Number of power iteration steps to be used. Increasing the number of power iterations improves accuracy, but lowers performance
i. **extra_samples** (*int, optional*) – Extra samples to be used besides the rank k. Can improve accuracy.
j. **dtype** (*type, optional*) – Enforces a type for elements of the decomposed matrix.

9. After training, apply the model to the BoW document, i.e. the 'corpus' variable and obtain **vecCorpus** = lsi[corpus]

10. Sort this vector obtained according to scores and select the top sentences

11. Output: sentences in decreasing order of their LSI scores

Summarization based on Non negative matrix factorization

1. Input: Text corpus (unlabeled)

2. Get the tokenized sentences using docSent

3. Create TF-IDF table of tokenized sentences

4. Vectorize the sentences using sentence tokens and the TF-IDF table

5. Now obtain the 'corpus' variable using **np.array**(vectorized sentences)

6. Train the NMF model and apply it to the 'corpus' to obtain 'Vectcorpus'. The tunable parameters are:
    a. **corpus** (iterable of list of (int, float) or csc_matrix with the shape (n_tokens, n_documents), optional) – Training corpus. Can be either iterable of documents, which are lists of (word_id, word_count), or a sparse csc matrix of BOWs for each document. If not specified, the model is left uninitialized (presumably, to be trained later with self.train()).
    b. **num_topics** (*int, optional*) – Number of topics to extract.
    c. **id2word** ({dict of (int, str), gensim.corpora.dictionary.Dictionary}) – Mapping from word IDs to words. It is used to determine the vocabulary size, as well as for debugging and topic printing.
    d. **chunksize** (*int, optional*) – Number of documents to be used in each training chunk.

    e. **passes** (*int, optional*) – Number of full passes over the training corpus. Leave at default passes=1 if your input is an iterator.
    f. **kappa** (*float, optional*) – Gradient descent step size. Larger value makes the model train faster, but could lead to non-convergence if set too large.
    g. **minimum_probability** – If normalize is True, topics with smaller probabilities are filtered out. If normalize is False, topics with smaller factors are filtered out. If set to None, a value of 1e-8 is used to prevent 0s.
    h. **w_max_iter** (*int, optional*) – Maximum number of iterations to train W per each batch.
    i. **w_stop_condition** (*float, optional*) – If error difference gets less than that, training of W stops for the current batch.
    j. **h_max_iter** (*int, optional*) – Maximum number of iterations to train h per each batch.
    k. **h_stop_condition** (*float*) – If error difference gets less than that, training of h stops for the current batch.
    l. **eval_every** (*int, optional*) – Number of batches after which l2 norm of (v - Wh) is computed. Decreases performance if set too low.
    m. **normalize** (*bool or None, optional*) – Whether to normalize the result. Allows for estimation of perplexity, coherence, e.t.c.
    n. **random_state** (*{np.random.RandomState, int}, optional*) – Seed for random generator. Needed for reproducibility.
    o.

7. Sort these 'Vectcorpus' on basis of the scores and take the top sentences

8. Output: Sentences in decreasing order of their NMF scores

9. Overall process very similar to LSI, only difference in factorization of the matrix

Summarization based on Page rank

1. Input: Text corpus (unlabeled)

2. Obtain the tokenized sentences from docSent

3. Then obtain the term frequency table (normal freq calculation)

4. Vectorize sentences using term frequency table

5. Arrange these vectors in form of numpy array to get 'sentvec'

6. simMat = cosine_similarity(sentVec)

7. Using similarity matrix construct a graph where nodes are sentences

8. Calculate ‘rank’ of these nodes using nx.pagerank
9. Sort sentences in descending order on basis of this rank
10. If diversification is needed then perform it using MMR otherwise just output sentences in descending order

Summarization based on Embedding page rank

1. Input: Text corpus (unlabeled)
2. Obtain tokenized sentences
3. Convert each sentence to ‘embeddings’ by adding the word2vec vectors of all the words in the sentence and then dividing by length of sentence for normalization
4. In this way sentence vectors are obtained
5. Convert these vectors to numpy array to get ‘sentVec’
6. simMat = cosine_similarity(sentVec)
7. Using similarity matrix construct a graph where nodes are sentences
8. Calculate ‘rank’ of these nodes using nx.pagerank
9. Sort sentences in descending order on basis of this rank
10. Output: top sentences on basis of the nx.pagerank score