Study Later-

Soft cosine similarity

LSA (latent semantic analysis)

Beautifulsoup

Which stemmer or lemmatizer to use?

Topic modelling (<https://towardsdatascience.com/understanding-feature-engineering-part-3-traditional-methods-for-text-data-f6f7d70acd41>)

Advanced pre-processing

1. Spelling corrections
2. Grammatical error correction
3. Removing repeated characters

Fasttext

<https://www.machinelearningplus.com/nlp/cosine-similarity/#:~:text=Euclidean%20distance%20instead.-,The%20cosine%20similarity%20is%20advantageous%20because%20even%20if%20the%20two,the%20angle%2C%20higher%20the%20similarity>.

Implement text classification in Python

<https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/>

dipanjan sarkar github

Question?

How to tune max\_idf/min\_idf in tfidfvectorizer?

How to tune parameters for tSNE? Do we need to keep default parameters?

TFIDFVectorizer in sklearn

1)

|  |
| --- |
| max\_df = 0.50 means "ignore terms that appear in **more than 50% of the documents**" |
| max\_df = 25 means "ignore terms that appear in **more than 25 documents**". |
| min\_df = 0.01 means "ignore terms that appear in **less than 1% of the documents**" |
| min\_df = 5 means "ignore terms that appear in less than 5 documents"   * TFidfVectorizer (in sklearn python) converts the text documents to a matrix of "tfidf" features   + tfidf features are the method to convert the textual information into the vector space, they are ameasure of how important a word in a text is * ngram is the number of words in a sequence   norm=’l2’: Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied.  2)  Here is a quick example that I could come up with to explain how the ngram range works in sklearn:  Example: range(1,2) v=text.CountVectorizer(ngram\_range=(1,2)) print(v.fit(["an apple a day keeps the doctor away"]).vocabulary\_)  Result: {'an': 0, 'apple': 2, 'day': 5, 'keeps': 9, 'the': 11, 'doctor': 7, 'away': 4, 'an apple': 1, 'apple day': 3, 'day keeps': 6, 'keeps the': 10, 'the doctor': 12, 'doctor away': 8}  Why is ‘a’ ignored in the result: because of the default token pattern in Count Vectorizer which ignores uni-character words. In a sense it is doing it correct because it has no information. You could alter this by passing the token pattern as an additional language.  v=CountVectorizer(ngram\_range=(1,2),token\_pattern='\b\w+\b')  3)  <https://www.analyticsvidhya.com/blog/2021/11/how-sklearns-tfidfvectorizer-calculates-tf-idf-values/>  <https://medium.com/analytics-vidhya/the-quantitative-value-of-text-tf-idf-and-more-e3c7883f1df3>  <https://stackoverflow.com/questions/47069744/is-smooth-idf-redundant>  tf is the number of times a term appears in a particular document. So it’s specific to a document. A few of the ways to calculate tf is given below:-  tf(t) = No. of times term ‘t’ occurs in a document  n= total number of documents  But as per sklearn’s online documentation, it uses the below method to calculate idf of a term in a document.  idf(t) = log e [ (1+n) / ( 1 + df(t) ) ] + 1 (default i:e smooth\_idf = True)  and  idf(t) = log e [ n / df(t) ] + 1 (when smooth\_idf = False)  tf-idf is highest for a t if it occurs many times within a small number of documents  tf-idf is lower for t when it occurs fewer times in a document, or occurs in many documents  tf-idf is lowest when t occurs in all the documents  It is not always true that multiple occurrences of a term in a document mean more significance of that term in proportion to the number of occurrences. Sublinear tf-scaling is modification of term frequency, which calculates weight as following,  https://miro.medium.com/max/520/1*w196eSpaeDKcHzCxY5gejA.png  Maximum tf-normalization, below  This is another modification of term frequency where tf of every term occurring in a document is normalized by the maximum tf in that document.  https://miro.medium.com/max/425/1*jbT8Olg0oAtet7qJPnoBTg.png  where *a* is the smoothing term ranging between 0 to 1 and is generally set to 0.4.  Maximum tf-normalization handles the case when a long document has higher values of term frequencies just because of the length of the document and it will have same terms repeated again and again.  This approach falls short in the case when a document will have a term occurring unusually very high number of times.  If smooth\_idf=True (the default), the constant “1” is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions: idf(d, t) = log [ (1 + n) / (1 + df(d, t)) ] + 1.  4)  If you go with the default values of tf-idf, meaning considering all terms, you have generated definitely more tokens. So your clustering process (or any other thing you want to do with those terms later) will take a longer time.  BUT the quality of your clustering should NOT be reduced.  One might think that allowing all terms (e.g. too frequent terms or stop-words) to be present might lower the quality but in tf-idf it doesn't. Because tf-idf measurement instinctively will give a low score to those terms, effectively making them not influential (as they appear in many documents).  So to sum it up, pruning the terms via min\_df and max\_df is to improve the performance, not the quality of clusters (as an example).  And the crucial point is that if you set the min and max mistakenly, you would lose some important terms and thus lower the quality. So if you are unsure about the right threshold (it depends on your documents set), or if you are sure about your machine's processing capabilities, leave the min, max parameters unchanged.  5)  Cosine similarity measures the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction. It is often used to measure document similarity in text analysis  6)  Cosine similarity is a metric used to determine how similar the documents are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. In this context, the two vectors I am talking about are arrays containing the word counts of two documents. As a similarity metric, how does cosine similarity differ from the number of common words? When plotted on a multi-dimensional space, where each dimension corresponds to a word in the document, the cosine similarity captures the orientation (the angle) of the documents and not the magnitude. If you want the magnitude, compute the Euclidean distance instead. The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance because of the size (like, the word ‘cricket’ appeared 50 times in one document and 10 times in another) they could still have a smaller angle between them. Smaller the angle, higher the similarity.  7)  A commonly used approach to match similar documents is based on counting the maximum number of common words between the documents. But this approach has an inherent flaw. That is, as the size of the document increases, the number of common words tend to increase even if the documents talk about different topics. The cosine similarity helps overcome this fundamental flaw in the ‘count-the-common-words’ or Euclidean distance approach.  8)  tf-idf, which stands for term frequency-inverse document frequency is similar to Bag of Words (BoW) where documents are considered as a bag or collection of words/terms and converted to numerical forms by counting the occurrences of every term. The whole idea is to assign a weight to each term occurring in the document.  **9) Limitations of PCA**  PCA is a linear algorithm. It will not be able to interpret complex polynomial relationship between features. On the other hand, t-SNE is based on probability distributions with random walk on neighborhood graphs to find the structure within the data.  10)  Chi-square explanation  <https://towardsdatascience.com/using-the-chi-squared-test-for-feature-selection-with-implementation-b15a4dad93f1>  11)  Can we use chi-square test (chi2) with tf-idf values?  Yes, because they are just the scaled values of the term frequency. In essence, chi-square calculates the sum of all the counts for a particular feature in a particular class, so essentially we can use the count values as feature for performing chi-square statistic.  Reference: <https://stackoverflow.com/questions/14573030/perform-chi-2-feature-selection-on-tf-and-tfidf-vectors>  12) t-sne  <https://www.youtube.com/watch?v=NEaUSP4YerM&t=8s>  <https://towardsdatascience.com/t-distributed-stochastic-neighbor-embedding-t-sne-bb60ff109561>  t-SNE is something called **nonlinear dimensionality reduction**. What that means is this algorithm allows us to separate data that cannot be separated by any straight line, let me show you an example:  Applies a non-linear dimensionality reduction technique where the focus is on keeping the very similar data points close together in lower-dimensional space.  t-SNE uses a heavy-tailed Student-t distribution to compute the similarity between two points in the low-dimensional space rather than a Gaussian distribution, which helps to address the crowding and optimization problems.  Outliers do not impact t-SNE  t-SNE converts the high-dimensional Euclidean distances between datapoints xᵢ and xⱼ into conditional probabilities P(j|i).  T- distribution creates the probability distribution of points in lower dimensions space, and this helps reduce the crowding issue.  **Find a low-dimensional data representation that minimizes the mismatch between Pᵢⱼ and qᵢⱼ using gradient descent based on Kullback-Leibler divergence(KL Divergence)**  When we minimize the KL divergence, it makes qᵢⱼ physically identical to Pᵢⱼ, so the structure of the data in high dimensional space will be similar to the structure of the data in low dimensional space.  Based on the KL divergence equation,  If Pᵢⱼ is large, then we need a large value for qᵢⱼ to represent local points with higher similarity.  If Pᵢⱼ is small, then we need a smaller value for qᵢⱼ to represent local points that are far apart.  [**https://medium.com/engineer-quant/t-sne-the-bits-that-no-one-learns-b5ce959ea1c2**](https://medium.com/engineer-quant/t-sne-the-bits-that-no-one-learns-b5ce959ea1c2)  However, before looking into t-SNE, we will investigate SNE, which uses Gaussian distribution instead of Student-t distriution. The first step is constructing the probability distribution of the dataset in high dimensions. The probabilities in the dataset are constructed using the Gaussian distribution which amounts to  https://miro.medium.com/max/436/1*N_CpCPqippE4uoa5mBRkIA.png  The intuition behind this construction is rather simple. The Euclidean distances between the pairwise points are considered to be normally distributed with mean of the point location and variance sigma, which the user defines.  Once the probability distribution in the high dimension is constructed, we construct a probability distribution in the lower dimensional space (usually 2 or 3 for visualization purposes) using the following formula for pairwise points y.  https://miro.medium.com/max/418/1*GCapOfi6dN0-bFxhVYWqOg.png  Now the goal is to make sure the two probability distributions are as similar as possible. This is achieved by considering the Kullback-Leibler (KL) divergence.  KL divergence is a measure of how different two probability distributions are from one another. The KL divergence is defined as  https://miro.medium.com/max/520/1*deqeY_4IiSUmAxYLHbZXbA.png  In essence, the KL divergence is the expected value of the log difference of the probabilities of the data points. Now that we have a notion of how to measure the similar/difference between distributions, we can optimize this by well known gradient descent methods.  However, there is a crucial disadvantage to using SNE. It is called the crowding problem and it is a result of the curse of dimensionality. Consider that you have some data points that have ten intrinsic dimensions, but represented in a much higher dimensional space and you want to reduce it to two dimensions. It is possible that if the number of data points is little, say 11, then there is no reliable way to map the data points in two dimensions.  Furthermore, when you map the high dimensional data in two dimensions, the area used by the data points in two dimensions that have high distances in the high dimensional data will not be sufficiently larger than the area used by the data points in two dimensions that have low distances in the high dimensional data. This is due to the fact that size scales up exponentially as dimensions increase. Consider the volume of sphere in n dimensions. It grows at a power of the radius. So, a sphere in n-dimensions will occupy much more space than a sphere in 2 dimensions, even with the same radius. So, the data in two dimensions tend to be ‘crowded’ together.  Since we are comparing the probabilities of the data points and not the actual distances, one natural way to overcome this issue is to consider distributions with fatter tails than Gaussian. One such distribution is the Student-t distribution. This works because when the tails are fatter, the distance of the points in the lower dimensional space is much bigger compared to when the tails are normally distributed. 13)2.2 Lexicon Normalization Another type of textual noise is about the multiple representations exhibited by single word.  For example – “play”, “player”, “played”, “plays” and “playing” are the different variations of the word – “play”, Though they mean different but contextually all are similar. The step converts all the disparities of a word into their normalized form (also known as lemma). Normalization is a pivotal step for feature engineering with text as it converts the high dimensional features (N different features) to the low dimensional space (1 feature), which is an ideal ask for any ML model.  The most common lexicon normalization practices are :   * **Stemming:** Stemming is a rudimentary rule-based process of stripping the suffixes (“ing”, “ly”, “es”, “s” etc) from a word. * **Lemmatization:**Lemmatization, on the other hand, is an organized & step by step procedure of obtaining the root form of the word, it makes use of vocabulary (dictionary importance of words) and morphological analysis (word structure and grammar relations).   14)  Word embeddings eventually help in establishing the association of a word with another similar meaning word through the created vectors.  15)  LSTM  <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>  16)  Word-preprocessing steps (<https://github.com/dipanjanS/practical-machine-learning-with-python/blob/master/notebooks/Ch07_Analyzing_Movie_Reviews_Sentiment/Text%20Normalization%20Demo.ipynb>)  <https://medium.com/m/global-identity?redirectUrl=https%3A%2F%2Ftowardsdatascience.com%2Funderstanding-feature-engineering-part-3-traditional-methods-for-text-data-f6f7d70acd41>   * Cleaning text (strip html) * Removing accented characters (e~ to e) * Expanding contractions (do not >> don’t ) * Lowercase * Insert space between special to isolate them (so that they can be removed later) * Lemmatize * Removing special characters * Remove extra white space * Removing stop words   17)  Disadvantage of BOW, TF-IDF based models  While they are effective methods for extracting features from text, due to the inherent nature of the model being just a bag of unstructured words, we lose additional information like the semantics, structure, sequence and context around nearby words in each text document.  18)  Word2vec-  <https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa>  <https://towardsdatascience.com/word-embeddings-exploration-explanation-and-exploitation-with-code-in-python-5dac99d5d795>  <https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>  19)  **CBOW Model:**This method takes the context of each word as the input and tries to predict the word corresponding to the context. Consider our example: Have a great day.  Let the input to the Neural Network be the word, great. Notice that here we are trying to predict a target word (day) using a single context input word great. More specifically, we use the one hot encoding of the input word and measure the output error compared to one hot encoding of the target word (day). In the process of predicting the target word, we learn the vector representation of the target word.  The input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons and the output is again a V length vector with the elements being the softmax values.  Let’s get the terms in the picture right: - Wvn is the weight matrix that maps the input x to the hidden layer (V\*N dimensional matrix)-W`nv is the weight matrix that maps the hidden layer outputs to the final output layer (N\*V dimensional matrix)  The hidden layer neurons just copy the weighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer.  The above model takes C context words. When Wvn is used to calculate hidden layer inputs, we take an average over all these C context word inputs.  **Skip-Gram:**  We input the target word into the network. The model outputs C probability distributions. What does this mean?  For each context position, we get C probability distributions of V probabilities, one for each word.  We have a pair of input words for each training example consisting of **one input target word** having a unique numeric identifier and **one context word** having a unique numeric identifier. If it is **a positive sample** the word has contextual meaning, is **a context word** and our **label Y=1**, else if it is a **negative sample**, the word has no contextual meaning, is just **a random word** and our **label Y=0**. We will pass each of them to an **embedding layer** of their own, having size **(vocab\_size x embed\_size)** which will give us **dense word embeddings** for each of these two words **(1 x embed\_size for each word)**. Next up we use a **merge layer** to compute the **dot product** of these two embeddings and get the dot product value. This is then sent to the **dense sigmoid layer** which outputs either a 1 or 0. We compare this with the actual label Y (1 or 0), compute the loss, backpropagate the errors to adjust the weights (in the embedding layer) and repeat this process for all **(target, context)** pairs for multiple epochs.  **Word2Vec vs Skip-Gram:**  Both have their own advantages and disadvantages. According to Mikolov, Skip Gram works well with small amount of data and is found to represent rare words well.  On the other hand, CBOW is faster and has better representations for more frequent words.  <https://ruder.io/word-embeddings-softmax/index.html#hierarchicalsoftmax>  **Hirearichal softmax in word2vec:-**  [**https://ruder.io/word-embeddings-softmax/index.html#hierarchicalsoftmax**](https://ruder.io/word-embeddings-softmax/index.html#hierarchicalsoftmax)  We can think of the regular softmax as a tree of depth 11, with each word in VV as a leaf node. Computing the softmax probability of one word then requires normalizing over the probabilities of all |V||V| leaves. If we instead structure the softmax as a binary tree, with the words as leaf nodes, then we only need to follow the path to the leaf node of that word, without having to consider any of the other nodes. Since a balanced binary tree has a depth of log2(|V|)log2(|V|), we only need to evaluate at most log2(|V|)log2(|V|) nodes to obtain the final probability of a word. Note that this probability is already normalized, as the probabilities of all leaves in a binary tree sum to 11 and thus form a probability distribution. To informally verify this, we can reason that at a tree's root node (Node 0) in Figure 1), the probabilities of branching decisions must sum to 11. At each subsequent node, the probability mass is then split among its children, until it eventually ends up at the leaf nodes, i.e. the words. Since no probability is lost along the way and since all words are leaves, the probabilities of all words must necessarily sum to 11 and hence the hierarchical softmax defines a normalized probability distribution over all words in VV.  **Negative Sampling (applied only to skip-gram models in word2vec):-**  [**https://stackoverflow.com/questions/27860652/word2vec-negative-sampling-in-layman-term**](https://stackoverflow.com/questions/27860652/word2vec-negative-sampling-in-layman-term)  The idea of word2vec is to maximise the similarity (dot product) between the vectors for words which appear close together (in the context of each other) in text, and minimise the similarity of words that do not. In equation (3) of the paper you link to, ignore the exponentiation for a moment. You have  v\_c . v\_w  -------------------  sum\_i(v\_ci . v\_w)  The numerator is basically the similarity between words c (the context) and w (the target) word. The denominator computes the similarity of all other contexts ci and the target word w. Maximising this ratio ensures words that appear closer together in text have more similar vectors than words that do not. However, computing this can be very slow, because there are many contexts ci. Negative sampling is one of the ways of addressing this problem- just select a couple of contexts ci at random. The end result is that if cat appears in the context of food, then the vector of food is more similar to the vector of cat (as measures by their dot product) than the vectors of **several other randomly chosen words** (e.g. democracy, greed, Freddy), instead of **all other words in language**. This makes word2vec much much faster to train.  The terminology is borrowed from classification, a common application of neural networks. There you have a bunch of positive and negative examples. With word2vec, for any given word you have a list of words that need to be similar to it (the positive class) but the negative class (words which are not similar to the targer word) is compiled by sampling.  ***# gensim’s Word2vec expects a sequence of sentences as its input,***  ***# where each sentence a list of words. We'll be lazy for now***  ***# and not perform any sort of text preprocessing***  20)  <http://ethen8181.github.io/machine-learning/deep_learning/word2vec/word2vec_detailed.html> *Negative Sampling* Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In other words, each training sample will tweak all of the weights in the neural network. As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples! Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here's how it works:  When training the network on the word pair ("fox", "quick"), we want the "correct output" of the network, that is the output neuron corresponding to "quick" to output a 1, and for all of the other thousands of output neurons to output a 0.  With negative sampling, we are instead going to randomly select just a small number of "negative" words (let's say 5) to update the weights for. (In this context, a "negative" word is one for which we want the network to output a 0 for). We will also still update the weights for our "positive" word (which is the word "quick" in our current example).  The paper says that selecting 5-20 words works well for smaller datasets, and we can get away with only 2-5 words for large datasets.  Recall that the output layer of our model has a weight matrix that's 300 x 10,000. So we will just be updating the weights for our positive word ("quick"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only 0.06% of the 3M weights in the output layer!  In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not). Selecting Negative Samples One little detail that's missing from the description above is how do we select the negative samples.The negative samples are chosen using the unigram distribution. Essentially, the probability of selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples. Instead of using the raw frequency for wiwi, freq(wi)freq(wi), in the original word2vec paper, each word is given a weight that's equal to it's frequency (word count) raised to the 3/4 power. The probability for selecting a word is just it's weight divided by the sum of weights for all words.  P(wi)=freq(wi)3/4∑nj=0(freq(wj)3/4)P(wi)=freq(wi)3/4∑j=0n(freq(wj)3/4)  This decision to raise the frequency to the 3/4 power appears to be empirical; as the author claims it outperformed other functions (e.g. just using unigram distribution).  Side note: The way this selection is implemented in the original word2vec C code is interesting. They have a large array with 100M elements (which they refer to as the unigram table). They fill this table with the index of each word in the vocabulary multiple times, and the number of times a word’s index appears in the table is given by P(wi)×table\_sizeP(wi)×table\_size. Then, to actually select a negative sample, we just generate a random integer between 0 and 100M, and use the word at that index in the table. Since the higher probability words occur more times in the table, we're more likely to pick those. *Subsampling Frequenct Words* Word2vec has two additional parameters for discarding some of the input words: words appearing less than **min-count** times are not considered as either words or contexts, and in addition frequent words are down-sampled as defined by the **sample** parameter.  There are two potential issues with frequently appeared words like "the":   * When looking at word pairs that includes "the", e.g. ("fox", "the"), "the" doesn't tell us much about the meaning of "fox", since it appears in the context of pretty much every word. * We will have more than enough samples of ("the", "the other word for the word pair") than we need to learn a good vector for "the".   Word2Vec implements a "subsampling" scheme to address this. For each word we encounter in our training text, there is a chance that we will discard it from the text. The probability that we cut the word is related to the word's frequency.  probability of keeping the word wi=(z(wi)0.001−−−−−√+1)⋅0.001z(wi)probability of keeping the word wi=(z(wi)0.001+1)⋅0.001z(wi)  Where:   * z(wi)z(wi) is the fraction of the total words in the corpus that are that word. For example, if the word "peanut" occurs 1,000 times in a 1 billion word corpus, then z("peanut") = 1E-6. * There is also a parameter called **sample** which controls how much subsampling occurs, and the default value is 0.001. Smaller values of **sample** mean words are less likely to be kept   Here are some interesting observations of this subsampling function (again this is using the default sample value of 0.001).   * probability of keeping the word wi=1probability of keeping the word wi=1 (100% chance of being kept) when z(wi)<=0.0026z(wi)<=0.0026. This means that only words which represent more than 0.26% of the total words will be subsampled * probability of keeping the word wi=0.5probability of keeping the word wi=0.5 (50% chance of being kept) when z(wi)<=0.00746z(wi)<=0.00746 * probability of keeping the word wi=0.033probability of keeping the word wi=0.033 (3.3% chance of being kept) when z(wi)=1.0z(wi)=1.0. That is, if the corpus consisted entirely of word wiwi, which of course is ridiculous   21) tokenizer.fit\_on\_texts and tokenizer.texts\_to\_sequences  <https://stackoverflow.com/questions/51956000/what-does-keras-tokenizer-method-exactly-do>  22) LSTM  <https://towardsdatascience.com/understanding-lstm-and-its-quick-implementation-in-keras-for-sentiment-analysis-af410fd85b47>  23) Attention mechanism  <https://medium.com/swlh/a-simple-overview-of-rnn-lstm-and-attention-mechanism-9e844763d07b>    24) **Grid vs Beam Search**  **https://towardsdatascience.com/foundations-of-nlp-explained-visually-beam-search-how-it-works-1586b9849a24#:~:text=Beam%20Search%20makes%20two%20improvements,considered%20each%20position%20in%20isolation.**   * With Greedy Search, we took just the single best word at each position. In contrast, Beam Search expands this and takes the best ’N’ words. * With Greedy Search, we considered each position in isolation. Once we had identified the best word for that position, we did not examine what came before it (ie. in the previous position), or after it. In contrast, Beam Search picks the ’N’ best *sequences*so far and considers the probabilities of the combination of all of the preceding words along with the word in the current position.   **Transformers**  <https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3> |
|  |