Topic modeling is an unsupervised machine learning technique that's capable of scanning a set of documents, detecting word and phrase patterns within them, and automatically clustering word groups and similar expressions that best characterize a set of documents.

Let's say I have found your diary (yeah, I know all the great hiding spots!) and I have only two minutes to understand your innermost secrets! How about reading it from the scratch? In two minutes? Nah, not possible! But I have a text mining robo-buddy who can process and analyze the whole diary in less than two minutes and through topic modeling, extract all much of the information out of it. Text mining techniques can quickly derive valuable knowledge and insights from large-scale (unstructured) text-based datasets such as books, journals, articles, speeches, digital documents and emails.

How Does Topic Modeling Work?

It's simple, really. Topic modeling involves counting words and grouping similar word patterns to infer topics within unstructured data. Let's say you're a software company and you want to know what customers are saying about particular features of your product. Instead of spending hours going through heaps of feedback, in an attempt to deduce which texts are talking about your topics of interest, you could analyze them with a topic modeling algorithm.

By detecting patterns such as word frequency and distance between words, a topic model clusters feedback that is similar, and words and expressions that appear most often. With this information, you can quickly deduce what each set of texts are talking about. Remember, this approach is 'unsupervised' meaning that no training is required.

LDA : Latent Dirichlet Allocation

It is one of the most popular topic modeling methods. Each document is made up of various words, and each topic also has various words belonging to it. The aim of LDA is to find topics a document belongs to, based on the words in it.

Example :

Model definition

```
Doc1: word1, word3, word5, word45, word11, word 62, word88 ...
Doc2: word9, word77, word31, word58, word83, word 92, word49 ...
Doc3: word44, word18, word52, word36, word64, word 11, word20 ...
Doc4: word85, word62, word19, word4, word30, word 94, word67 ...
Doc5: word19, word53, word74, word79, word45, word 39, word54 ...
```

Each document is a collection of words.

We have 5 documents each containing the words listed in front of them(ordered by frequency of occurrence).

What we want to figure out are the words in different topics, as shown in the table below. Each row in the table represents a different topic and each column a different word in the corpus. Each cell contains the probability that the word(column) belongs to the topic(row).

	Word1	word2	word3	word4	.....
Topic1	0.01	0.23	0.19	0.03	
Topic2	0.21	0.07	0.48	0.02	
Topic3	0.53	0.01	0.17	0.04	

Each topic contains a score for all the words in the corpus.

Finding Representative Words for a Topic

- We can **sort the words** with respect to their probability score. The top x words are chosen from each topic to represent the

topic. If $x = 10$, we'll sort all the words in topic1 based on their score and take the top 10 words to represent the topic.
This step may not always be necessary because if the corpus is small we can store all the words in sorted by their score.

- Alternatively, we can **set a threshold** on the score. All the words in a topic having a score above the threshold can be stored as its representative, in order of their scores.

Let's say we have 2 topics that can be classified as *CAT_related* and *DOG_related*. A topic has probabilities for each word, so words such as *milk*, *meow*, and *kitten*, will have a higher probability in the *CAT_related* topic than in the *DOG_related* one. The *DOG_related* topic, likewise, will have high probabilities for words such as *puppy*, *bark*, and *bone*.

If we have a document containing the following sentences:

"*Dogs* like to *chew* on *bones* and fetch sticks".
"*Puppies* drink *milk*."
"Both like to *bark*."

We can easily say it belongs to topic *DOG_related* because it contains words *such as Dogs*, *bones, puppies*, and *bark*. Even though it contains the word *milk* which belongs to the topic *CAT_related*, the document belongs to *DOG_related* as more words match with it.

Assumptions :

- Each document is just a collection of words or a “bag of words”. Thus, the **order of the words** and the **grammatical role** of the words (subject, object, verbs, …) are **not considered** in the model.
- Words like am/is/are/of/a/the/but/… don’t carry any information about the “topics” and therefore can be eliminated from the documents as a preprocessing step. In fact, **we can eliminate words that occur in at least %80 ~ %90 of the documents,** without losing any information.
  For example, if our corpus contains only medical documents, words like human, body, health, etc might be present in most of the documents and hence can be removed as they don’t add any specific information which would make the document stand out.
- We **know beforehand how many topics** we want. ‘***k***’ is pre-decided.
- **All topic assignments except for the current word in question are correct**, and then updating the assignment of the current word using our model of how documents are generated

How does LDA work?

There are 2 parts in LDA:

- The ***words that belong to a document***, that we already know.
- The ***words that belong to a topic*** or the probability of words belonging into a topic, that we need to calculate.

The Algorithm to find the latter

- Go through each document and randomly assign each word in the document to one of ***k*** topics (*k* is chosen beforehand).
- For each document ***d***, go through each word ***w*** and compute :

1. **p(topic *t* | document *d*)**: the **proportion of words in document *d* that are assigned to topic *t***. Tries to capture how many words belong to the topic *t* for a given document *d*. Excluding the current word.
   If a lot of words from *d* belongs to *t*, it is more probable that word *w* belongs to *t*.
   (#words in *d* with *t* +*alpha*/ #words in *d* with any topic+ *k***alpha*)
2. **p(word *w*| topic *t*)**: the proportion of assignments to topic *t* over all documents that come from this word *w*. Tries to capture how many documents are in topic *t* because of word *w*. LDA represents documents as a mixture of topics. Similarly, a topic is a mixture of words. If a word has high probability of being in a topic, all the documents having *w* will be more strongly associated with *t* as well. Similarly, if *w* is not very probable to be in *t*, the documents which contain the *w* will be having very low probability of being in *t,* because rest of the words in *d* will belong to some other topic and hence *d* will have a higher probability for those topic. So even if *w* gets added to *t*, it won't be bringing many such documents to *t*.

- Update the probability for the word *w* belonging to topic *t*, as

```
p(word w with topic t) =
p(topic t | document d) * p(word w | topic t)
```

A layman's example

Suppose you have various photographs(*documents*) with captions(*words*). You want to display them in a gallery so you decide to categorize the photographs on various themes(*topics*) based on which you will create different sections in your gallery.

You decide to create *k=2* sections in your album — nature and city. Naturally, the classification isn't so clear as some photographs with city have trees and flowers while the nature ones might have some buildings in it. You, as a start, decide to assign the photographs which only have nature or city elements in them into their respective categories while you randomly assigned the rest.

You notice that a lot of photographs in nature have the word *tree* in their captions. So you concluded that the word *tree* and topic *nature* must be closely related.

Next, you pick the word *building* and check how many photographs are in *nature* because they have the word *building* in their caption. You don't find many and now are less sure about *building* belonging to the topic *nature* and associate it more strongly with the topic *city*.

You then pick a photograph which has the caption **"The tree is in front of the building and behind a car"** and see that it is in the category nature currently.
You then chose the word ***tree***, and calculate the first probability **p(topic *t* | document *d*):** other words in the caption are *building* and *car*, most photographs having captions with *building* or *car* in it are in *city*, so you get a low probability.
Now the second probability **p(word *w*| topic *t*)**: we know that a lot of photographs in nature have the word *trees* in it. So you get a high score here.
You update the probability of *tree* belonging in *nature* by multiplying the two. You get a lower value than before for *tree* in *topic* nature because now you have seen that *tree* and words such as *building*/*car* in the same caption, implying that trees can also be found in cities.
For the same reason, when you update the probability for *tree* belonging in topic *city*, you will notice that it will be greater than what it was before.

After multiple iterations over all the photographs and for each topic, you will have accurate scores for each word with respect to each topic. Your guesses will keep getting better and better because you'll conclude from the context that words such as buildings, sidewalk, subway appear together and hence must belong to the same topic, which we can easily guess is *city*.
Words such as mountains, fields, beach which might not appear together in a lot of captions but they do appear often without city words and hence will have higher scores for nature.

While words such as trees, flowers, dogs, sky will have almost the same probability of being in either as they occur in both topics.

As for the photograph, you see that it has 1 word (with average probability) from category nature and 2 words (with high probability) from city, you conclude, it belongs to city more strongly than it does to nature and hence you decide to add it in city.

Side note

The applications of LDA need not be restricted to Natural Language Processing. I recently (Ria Kulshrestha) implemented a paper where we use LDA (along with a Neural Networks) to extract the scene-specific context of an image.

LDA section is published by Ria Kulshrestha in Towards Data Science:
https://towardsdatascience.com/latent-dirichlet-allocation-lda-9d1cd064ffa2

Data preprocessing for LDA

The typical preprocessing steps before performing LDA are
1) tokenization,
2) punctuation and special character removal,
3) stop word removal and
4) lemmatized.

Note that additional preprocessing may be required based on the quality of the data.

1. **Tokenization**: getting the normalized text and splitting it into a list of tokens. In Natural Language Processing, String Tokenization is a process where the string is splitted into Individual words or Individual parts without blanks and tabs. In the same step, the words in the String is converted into lower case. The Tokenize Module from NLTK or Naural Language Toolkit makes very easy to carry out this process.

```
import nltk
from nltk.tokenize import (word_tokenize,
                           sent_tokenize,
                           TreebankWordTokenizer,
                           wordpunct_tokenize,
                           TweetTokenizer,
                           MWETokenizer)
text="Hope, is the only thing stronger than fear! #Hope #Amal.M"
```

- Word and Sentence tokenizer

```
print(word_tokenize(text))
```

```
['Hope', ',', 'is', 'the', 'only', 'thing', 'stronger', 'than', 'fear', '!', '#', 'Hope', '#', 'Ama
```

```
print(sent_tokenize(text))
```

```
['Hope, is the only thing stronger than fear!', '#Hope #Amal.M']
```

N.B: The sent_tokenize uses the pre-trained model from tokenizers/punkt/english.pickle.

2. **Punctuation**: is the system of symbols that we use to separate written sentences and parts of sentences, and to make their meaning clear. Each symbol is called a "punctuation mark".

```
# remove punctuation from text and also word that have a length small or equal to 3

def clean_text(text ):
    delete_dict = {sp_character: '' for sp_character in string.punctuation}
    delete_dict[' '] = ' '
    table = str.maketrans(delete_dict)
    text1 = text.translate(table)
    #print('cleaned:'+text1)
    textArr= text1.split()
    text2 = ' '.join([w for w in textArr if ( not w.isdigit() and  ( not w.isdigit() and len(w)>3))])

    return text2.lower()
```

3. **Removing stop words**: stop words are the words that are most commonly used in a language and do not add much meaning to the text. Some examples are the words ‘the’, ‘a’, ‘will’, …

```
# remove stopwords
stop_words = stopwords.words('english')
# function to remove stopwords
def remove_stopwords(text):
    textArr = text.split(' ')
    rem_text = " ".join([i for i in textArr if i not in stop_words])
    return rem_text
```

4. **Lemmatization**: This is the process of getting the same word for a group of inflected word forms, the simplest way to do this is with a dictionary. Examples: is, was, were become be.

```
nlp = spacy.load('en_core_web_sm', disable=['parser', 'ner'])
```

```
def lemmatization(texts,allowed_postags=['NOUN', 'ADJ']):
       output = []
       for sent in texts:
             doc = nlp(sent)
             output.append([token.lemma_ for token in doc if token.pos_ in allowed_postags ])
       return output
```