Gensim Tutorial – A Complete Beginners Guide

by Selva Prabhakaran (https://www.machinelearningplus.com/author/selva86/) | Posted on October 16, 2018 (https://www.machinelearningplus.com/nlp/gensim-tutorial/)

Gensim is billed as a Natural Language Processing package that does 'Topic Modeling for Humans'. But it is practically much more than that. It is a leading and a state-of-the-art package for processing texts, working with word vector models (such as Word2Vec, FastText etc) and for building topic models.

Gensim Tutorial – A Complete Beginners Guide. Photo by Jasmin Schreiber (https://unsplash.com/photos/Fpi3B9RMe5E?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

[container] [columnize] 1. Introduction

Contents

[/columnize] [/container]

1. Introduction

What is gensim?

Gensim is billed as a Natural Language Processing package that does 'Topic Modeling for Humans'. But its practically much more than that.

If you are unfamiliar with topic modeling (https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/), it is a technique to extract the underlying topics from large volumes of text. Gensim provides algorithms like LDA and LSI (which we will see later in this post) and the necessary sophistication to build high-quality topic models.

You may argue that topic models and word embedding are available in other packages like scikit, R etc. But the width and scope of facilities to build and evaluate topic models are unparalleled in gensim, plus many more convenient facilities for text processing.

It is a great package for processing texts, working with word vector models (such as Word2Vec, FastText etc) and for building topic models.

Also, another significant advantage with gensim is: it lets you handle large text files without having to load the entire file in memory.

This post intends to give a practical overview of the nearly all major features, explained in a simple and easy to understand way.

By the end of this tutorial, you would know:

- What are the core concepts in gensim?
- What is dictionary and corpus, why they matter and where to use them?
- How to create and work with dictionary and corpus?
- How to load and work with text data from multiple text files in memory efficient way
- Create topic models with LDA and interpret the outputs
- Create TFIDF model, bigrams, trigrams, Word2Vec model, Doc2Vec model
- Compute similarity metrics
- And much more..

Let's begin.

2. What is a Dictionary and Corpus?

In order to work on text documents, Gensim requires the words (aka tokens) be converted to unique ids. In order to achieve that, Gensim lets you create a `Dictionary` object that maps each word to a unique id.

So, how to create a `Dictionary`? By converting your text/sentences to a [list of words] and pass it to the `corpora.Dictionary()` object.

We will see how to actually do this in the next section.

Matplotlib Histogram – How to Visualize Distributions in Python (https://www.machinelearningplus.com/plots/matplot histogram-python-examples/)

ARIMA Model – Complete Guide to Time Series Forecasting in Python (https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/)

Time Series Analysis in Python – A Comprehensive Guide with Examples (https://www.machinelearningplus.com/time-series/time-series-analysis-python/)

Matplotlib Tutorial – A Complete Guide to Python Plot w/ Examples (https://www.machinelearningplus.com/plots/matplot tutorial-complete-guide-python-plot-examples/)

Topic modeling visualization – How to present the results of LDA models? (https://www.machinelearningplus.com/nlp/topic-modeling-visualization-how-to-present-results-lda-models/)

Top 50 matplotlib Visualizations – The Master Plots (with full python code) (https://www.machinelearningplus.com/plots/top-50-matplotlib-visualizations-the-master-plots-python/)

List Comprehensions in Python – My Simplified Guide (https://www.machinelearningplus.com/python/list-comprehensions-in-python/)

Python @Property Explained – How to Use and When? (Full Examples) (https://www.machinelearningplus.com/python/pytho property/)

How Naive Bayes Algorithm Works? (with example and full code) (https://www.machinelearningplus.com/predictive-modeling/how-naive-bayes-algorithm-works-with-example-and-full-code/)

Tags

Bigrams (https://www.machinelearningplus.com/tag/bigrams/) Classification (https://www.machinelearningplus.com/tag/cla Corpus (https://www.machinelearningplus.com/tag/corpus/) Cosine Similarity (https://www.machinelearningplus.com/tag/cosine-similarity/) data.table (https://www.machinelearningplus.com/tag/data-table/) Data Manipulation (https://www.machinelearningplus.com/tag manipulation/) Debugging (https://www.machinelearningplus.com/tag/debugging/) Doc2Vec (https://www.machinelearningplus.com/tag/doc2vec/)

But why is the dictionary object needed and where can it be used?

The dictionary object is typically used to create a 'bag of words' Corpus. It is this Dictionary and the bag-of-words (Corpus) that are used as inputs to topic modeling and other models that Gensim specializes in.

Alright, what sort of text inputs can gensim handle? The input text typically comes in 3 different forms:

1. As sentences stored in python's native list object
2. As one single text file, small or large.
3. In multiple text files.

Now, when your text input is large, you need to be able to create the dictionary object without having to load the entire text file.

The good news is Gensim lets you read the text and update the dictionary, one line at a time, without loading the entire text file into system memory. Let's see how to do that in the next 2 sections.

But, before we get in, let's understand some NLP jargon.

A 'token' typically means a 'word'. A 'document' can typically refer to a 'sentence' or 'paragraph' and a 'corpus' is typically a 'collection of documents as a bag of words'. That is, for each document, a corpus contains each word's id and its frequency count in that document. As a result, information of the order of words is lost.

If everything is clear so far, let's get our hands wet and see how to create the dictionary from a list of sentences.

3. How to create a Dictionary from a list of sentences?

In gensim, the dictionary contains a map of all words (tokens) to its unique id.

You can create a dictionary from a paragraph of sentences, from a text file that contains multiple lines of text and from multiple such text files contained in a directory. For the second and third cases, we will do it without loading the entire file into memory so that the dictionary gets updated as you read the text line by line.

Let's start with the 'List of sentences' input.

When you have multiple sentences, you need to convert each sentence to a list of words. List comprehensions is a common way to do this.

```
import gensim
from gensim import corpora
from pprint import pprint

# How to create a dictionary from a list of sentences?
documents = ["The Saudis are preparing a report that will acknowledge that",
             "Saudi journalist Jamal Khashoggi's death was the result of an",
             "interrogation that went wrong, one that was intended to lead",
             "to his abduction from Turkey, according to two sources."]

documents_2 = ["One source says the report will likely conclude that",
                "the operation was carried out without clearance and",
                "transparency and that those involved will be held",
                "responsible. One of the sources acknowledged that the",
                "report is still being prepared and cautioned that",
                "things could change."]

# Tokenize(split) the sentences into words
texts = [[text for text in doc.split()] for doc in documents]

# Create dictionary
dictionary = corpora.Dictionary(texts)

# Get information about the dictionary
print(dictionary)
#> Dictionary(33 unique tokens: ['Saudis', 'The', 'a', 'acknowledge', 'are']...)
```

As it says the dictionary has 34 unique tokens (or words). Let's see the unique ids for each of these tokens.

```
# Show the word to id map
print(dictionary.token2id)
#> {'Saudis': 0, 'The': 1, 'a': 2, 'acknowledge': 3, 'are': 4,
#> 'preparing': 5, 'report': 6, 'that': 7, 'will': 8, 'Jamal': 9,
#> "Khashoggi's": 10, 'Saudi': 11, 'an': 12, 'death': 13,
#> 'journalist': 14, 'of': 15, 'result': 16, 'the': 17, 'was': 18,
#> 'intended': 19, 'interrogation': 20, 'lead': 21, 'one': 22,
#> 'to': 23, 'went': 24, 'wrong,': 25, 'Turkey,': 26, 'abduction': 27,
#> 'according': 28, 'from': 29, 'his': 30, 'sources.': 31, 'two': 32}
```

We have successfully created a Dictionary object. Gensim will use this dictionary to create a bag-of-words corpus where the words in the documents are replaced with its respective id provided by this dictionary.

If you get new documents in the future, it is also possible to update an existing dictionary to include the new words.

series/) Topic Modeling (https://www.machinelearningplus.com/tag/topic-modeling/) Visualization (https://www.machinelearningplus.com/tag/visualiz
Word2Vec (https://www.machinelearningplus.com/tag/word2vec/)

```
documents_2 = ["The intersection graph of paths in trees",
               "Graph minors IV Widths of trees and well quasi ordering",
               "Graph minors A survey"]

texts_2 = [[text for text in doc.split()] for doc in documents_2]

dictionary.add_documents(texts_2)



# If you check now, the dictionary should have been updated with the new words (tokens).
print(dictionary)
#> Dictionary(45 unique tokens: ['Human', 'abc', 'applications', 'computer', 'for']...)

print(dictionary.token2id)
#> {'Human': 0, 'abc': 1, 'applications': 2, 'computer': 3, 'for': 4, 'interface': 5,
#>  'lab': 6, 'machine': 7, 'A': 8, 'of': 9, 'opinion': 10, 'response': 11, 'survey': 12,
#>  'system': 13, 'time': 14, 'user': 15, 'EPS': 16, 'The': 17, 'management': 18,
#>  'System': 19, 'and': 20, 'engineering': 21, 'human': 22, 'testing': 23, 'Relation': 24,
#>  'error': 25, 'measurement': 26, 'perceived': 27, 'to': 28, 'binary': 29, 'generation': 30,
#>  'random': 31, 'trees': 32, 'unordered': 33, 'graph': 34, 'in': 35, 'intersection': 36,
#>  'paths': 37, 'Graph': 38, 'IV': 39, 'Widths': 40, 'minors': 41, 'ordering': 42,
#>  'quasi': 43, 'well': 44}
```

4. How to create a Dictionary from one or more text files?

You can also create a dictionary from a text file or from a directory of text files.

The below example reads a file line-by-line and uses gensim's simple_preprocess to process one line of the file at a time.

The advantage here is it let's you read an entire text file without loading the file in memory all at once.

Let's use a sample.txt (https://www.machinelearningplus.com/wp-content/uploads/2018/10/sample.txt) file to demonstrate this.

```
 from gensim.utils import simple_preprocess
from smart_open import smart_open
import os

# Create gensim dictionary form a single tet file
dictionary = corpora.Dictionary(simple_preprocess(line, deacc=True) for line in open('sample.tx

# Token to Id map
dictionary.token2id

#> {'according': 35,
#>  'and': 22,
#>  'appointment': 23,
#>  'army': 0,
#>  'as': 43,
#>  'at': 24,
#>   ...
#> }
```

We have created a dictionary from a single text file. Nice!

Now, how to read one-line-at-a-time from *multiple* files?

Assuming you have all the text files in the same directory, you need to define a class with an `__iter__` method. The `__iter__()` method should iterate through all the files in a given directory and yield the processed list of word tokens.

Let's define one such class by the name `ReadTxtFiles`, which takes in the path to directory containing the text files. I am using this directory of sports food docs (https://github.com/selva86/datasets/tree/master/lsa_sports_food_docs) as input.

```
class ReadTxtFiles(object):
    def __init__(self, dirname):
        self.dirname = dirname

    def __iter__(self):
        for fname in os.listdir(self.dirname):
            for line in open(os.path.join(self.dirname, fname), encoding='latin'):
                yield simple_preprocess(line)

path_to_text_directory = "lsa_sports_food_docs"

dictionary = corpora.Dictionary(ReadTxtFiles(path_to_text_directory))

# Token to Id map
dictionary.token2id
# {'across': 0,
#  'activity': 1,
#  'although': 2,
#  'and': 3,
#  'are': 4,
#  ...
# }
```

This blog_post (https://rare-technologies.com/data-streaming-in-python-generators-iterators-iterables/) gives a nice overview to understand the concept of iterators and generators.

5. How to create a bag of words corpus in gensim?

Now you know how to create a dictionary from a list and from text file.

The next important object you need to familiarize with in order to work in gensim is the Corpus (a Bag of Words). That is, it is a corpus object that contains the word id and its frequency in each document. You can think of it as gensim's equivalent of a Document-Term matrix.

Once you have the updated dictionary, all you need to do to create a bag of words corpus is to pass the tokenized list of words to the `Dictionary.doc2bow()`

Let's create s Corpus for a simple list (`my_docs`) containing 2 sentences.

```
# List with 2 sentences
my_docs = ["Who let the dogs out?",
           "Who? Who? Who? Who?"]

# Tokenize the docs
tokenized_list = [simple_preprocess(doc) for doc in my_docs]

# Create the Corpus
mydict = corpora.Dictionary()
mycorpus = [mydict.doc2bow(doc, allow_update=True) for doc in tokenized_list]
pprint(mycorpus)
#> [[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1)], [(4, 4)]]
```

How to interpret the above corpus?

The (0, 1) in line 1 means, the word with id=0 appears once in the 1st document. Likewise, the (4, 4) in the second list item means the word with id 4 appears 4 times in the second document. And so on.

Well, this is not human readable. To convert the id's to words, you will need the dictionary to do the conversion.

Let's see how to get the original texts back.

```
word_counts = [[(mydict[id], count) for id, count in line] for line in mycorpus]
pprint(word_counts)
#> [[('dogs', 1), ('let', 1), ('out', 1), ('the', 1), ('who', 1)], [('who', 4)]]
```

Notice, the order of the words gets lost. Just the word and it's frequency information is retained.

6. How to create a bag of words corpus from a text file?

Reading words from a python list is quite straightforward because the entire text was in-memory already.
However, you may have a large file that you don't want to load the entire file in memory.

You can import such files one line at a time by defining a class and the `__iter__` function that iteratively reads the file one line at a time and yields a corpus object. But how to create the corpus object?

The `__iter__()` from `BoWCorpus` reads a line from the file, process it to a list of words using `simple_preprocess()` and pass that to the `dictionary.doc2bow()`. Can you related how this is similar and different from the `ReadTxtFiles` class we created earlier?

Also, notice that I am using the `smart_open()` from smart_open (https://github.com/RaRe-Technologies/smart_open) package because, it lets you open and read large files line-by-line from a variety of sources such as S3, HDFS, WebHDFS, HTTP, or local and compressed files. That's pretty awesome by the way!

However, if you had used `open()` for a file in your system, it will work perfectly file as well.

```
 from gensim.utils import simple_preprocess
from smart_open import smart_open
import nltk
nltk.download('stopwords')  # run once
from nltk.corpus import stopwords
stop_words = stopwords.words('english')



class BoWCorpus(object):
    def __init__(self, path, dictionary):
        self.filepath = path
        self.dictionary = dictionary

    def __iter__(self):
        global mydict  # OPTIONAL, only if updating the source dictionary.
        for line in smart_open(self.filepath, encoding='latin'):
            # tokenize
            tokenized_list = simple_preprocess(line, deacc=True)

            # create bag of words
            bow = self.dictionary.doc2bow(tokenized_list, allow_update=True)

            # update the source dictionary (OPTIONAL)
            mydict.merge_with(self.dictionary)

            # lazy return the BoW
            yield bow



# Create the Dictionary
mydict = corpora.Dictionary()

# Create the Corpus
bow_corpus = BoWCorpus('sample.txt', dictionary=mydict)  # memory friendly

# Print the token_id and count for each line.
for line in bow_corpus:
    print(line)

#> [(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1), (1
#> [(12, 1), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1)]
#> ... truncated ...
```

7. How to save a gensim dictionary and corpus to disk and load them back?

This is quite straightforward. See the examples below.

```
 # Save the Dict and Corpus
mydict.save('mydict.dict')  # save dict to disk
corpora.MmCorpus.serialize('bow_corpus.mm', bow_corpus)  # save corpus to disk
```

We have saved the dictionary and corpus objects. Let's load them back.

```
# Load them back
loaded_dict = corpora.Dictionary.load('mydict.dict')


corpus = corpora.MmCorpus('bow_corpus.mm')
for line in corpus:
    print(line)
```

8. How to create the TFIDF matrix (corpus) in gensim?

The Term Frequency – Inverse Document Frequency(TF-IDF) is also a bag-of-words model but unlike the regular corpus, TFIDF down weights tokens (words) that appears frequently across documents.

How is TFIDF computed?

Tf-Idf is computed by multiplying a local component like term frequency (TF) with a global component, that is, inverse document frequency (IDF) and optionally normalizing the result to unit length.

As a result of this, the words that occur frequently across documents will get downweighted.

There are multiple variations of formulas for TF and IDF existing. Gensim uses the SMART Information retrieval system (https://en.wikipedia.org/wiki/SMART_Information_Retrieval_System) that can be used to implement these variations. You can specify what formula to use specifying the `smartirs` parameter in the `TfidfModel`. See `help(models.TfidfModel)` for more details.

So, how to get the TFIDF weights?

By training the corpus with `models.TfidfModel()`. Then, apply the corpus within the square brackets of the trained `tfidf` model. See example below.

```
 from gensim import models
import numpy as np

documents = ["This is the first line",
             "This is the second sentence",
             "This third document"]

# Create the Dictionary and Corpus
mydict = corpora.Dictionary([simple_preprocess(line) for line in documents])
corpus = [mydict.doc2bow(simple_preprocess(line)) for line in documents]

# Show the Word Weights in Corpus
for doc in corpus:
    print([[mydict[id], freq] for id, freq in doc])

# [['first', 1], ['is', 1], ['line', 1], ['the', 1], ['this', 1]]
# [['is', 1], ['the', 1], ['this', 1], ['second', 1], ['sentence', 1]]
# [['this', 1], ['document', 1], ['third', 1]]

# Create the TF-IDF model
tfidf = models.TfidfModel(corpus, smartirs='ntc')

# Show the TF-IDF weights
for doc in tfidf[corpus]:
    print([[mydict[id], np.around(freq, decimals=2)] for id, freq in doc])
# [['first', 0.66], ['is', 0.24], ['line', 0.66], ['the', 0.24]]
# [['is', 0.24], ['the', 0.24], ['second', 0.66], ['sentence', 0.66]]
# [['document', 0.71], ['third', 0.71]]
```

Notice the difference in weights of the words between the original corpus and the tfidf weighted corpus.

The words 'is' and 'the' occur in two documents and were weighted down. The word 'this' appearing in all three documents was removed altogether. In simple terms, words that occur more frequently across the documents get smaller weights.

9. How to use gensim downloader API to load datasets?

Gensim provides an inbuilt API to download popular text datasets and word embedding models.

A comprehensive list of available datasets and models is maintained here (https://raw.githubusercontent.com/RaRe-Technologies/gensim-data/master/list.json).

Using the API to download the dataset is as simple as calling the `api.load()` method with the right data or model name.

The below example shows how to download the 'glove-wiki-gigaword-50' model.

```
import gensim.downloader as api

# Get information about the model or dataset
api.info('glove-wiki-gigaword-50')
# {'base_dataset': 'Wikipedia 2014 + Gigaword 5 (6B tokens, uncased)',
#  'checksum': 'c289bc5d7f2f02c6dc9f2f9b67641813',
#  'description': 'Pre-trained vectors based on Wikipedia 2014 + Gigaword, 5.6B tokens, 400K vo
#  'file_name': 'glove-wiki-gigaword-50.gz',
#  'file_size': 69182535,
#  'license': 'http://opendatacommons.org/licenses/pddl/ (http://opendatacommons
#  (... truncated...)

# Download
w2v_model = api.load("glove-wiki-gigaword-50")
w2v_model.most_similar('blue')
# [('red', 0.8901656866073608),
#  ('black', 0.8648407459259033),
#  ('pink', 0.8452916741371155),
#  ('green', 0.8346816301345825),
#  ... ]
```

10. How to create bigrams and trigrams using Phraser models?

Now you know how to download datasets and pre-trained models with gensim.

Let's download the `text8` dataset, which is nothing but the "First 100,000,000 bytes of plain text from Wikipedia". Then, from this, we will generate bigrams and trigrams.

But what are bigrams and trigrams? and why do they matter?

In paragraphs, certain words always tend to occur in pairs (bigram) or in groups of threes (trigram). Because the two words combined together form the actual entity. For example: The word 'French' refers the language or region and the word 'revolution' can refer to the planetary revolution. But combining them, 'French Revolution', refers to something completely different.

It's quite important to form bigrams and trigrams from sentences, especially when working with bag-of-words models.

So how to create the bigrams?

It's quite easy and efficient with gensim's `Phrases` model. The created `Phrases` model allows indexing, so, just pass the original text (list) to the built `Phrases` model to form the bigrams. An example is shown below:

```
 dataset = api.load("text8")
dataset = [wd for wd in dataset]


dct = corpora.Dictionary(dataset)
corpus = [dct.doc2bow(line) for line in dataset]


# Build the bigram models
bigram = gensim.models.phrases.Phrases(dataset, min_count=3, threshold=10)


# Construct bigram
print(bigram[dataset[0]])
# ['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used',
#  'against', 'early', 'working_class', 'radicals', 'including', 'the', 'diggers',
#  'of', 'the', 'english', 'revolution', 'and', 'the', 'sans_culottes', 'of', 'the',
#  'french_revolution', 'whilst',...]
```

The bigrams are ready. Can you guess how to create a trigram?

Well, Simply rinse and repeat the same procedure to the output of the bigram model. Once you've generated the bigrams, you can pass the output to train a new `Phrases` model. Then, apply the bigrammed corpus on the trained trigram model. Confused? See the example below.

```
 # Build the trigram models
trigram = gensim.models.phrases.Phrases(bigram[dataset], threshold=10)


# Construct trigram
print(trigram[bigram[dataset[0]]])
```

11. How to create Topic Models with LDA?

The objective of topic models is to extract the underlying topics from a given collection of text documents. Each document in the text is considered as a combination of topics and each topic is considered as a combination of related words.

Topic modeling can be done by algorithms like Latent Dirichlet Allocation (LDA) and Latent Semantic Indexing (LSI).

In both cases you need to provide the number of topics as input. The topic model, in turn, will provide the topic keywords for each topic and the percentage contribution of topics in each document.

The quality of topics is highly dependent on the quality of text processing and the number of topics you provide to the algorithm. The earlier post on how to build best topic models explains the procedure in more detail. However, I recommend understanding the basic steps involved and the interpretation in the example below.

Step 0: Load the necessary packages and import the stopwords.

```
# Step 0: Import packages and stopwords
from gensim.models import LdaModel, LdaMulticore
import gensim.downloader as api
from gensim.utils import simple_preprocess, lemmatize
from nltk.corpus import stopwords
import re
import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s')
logging.root.setLevel(level=logging.INFO)
stop_words = stopwords.words('english')
stop_words = stop_words + ['com', 'edu', 'subject', 'lines', 'organization', 'would', 'article'
```

Step 1: Import the dataset. I am going to use the `text8` dataset that can be downloaded using gensim's downloader API.

```
# Step 1: Import the dataset and get the text and real topic of each news article
dataset = api.load("text8")
data = [d for d in dataset]
```

Step 2: Prepare the downloaded data by removing stopwords and lemmatize (https://www.machinelearningplus.com/nlp/lemmatization-examples-python/) it. For Lemmatization, gensim requires the `pattern` package. So, be sure to do `pip install pattern` in your terminal or prompt before running this. I have setup lemmatization such that only Nouns (NN), Adjectives (JJ) and Pronouns (RB) are retained. Because I prefer only such words to go as topic keywords. This is a personal choice.

```
# Step 2: Prepare Data (Remove stopwords and lemmatize)
data_processed = []

for i, doc in enumerate(data[:100]):
    doc_out = []
    for wd in doc:
        if wd not in stop_words:  # remove stopwords
            lemmatized_word = lemmatize(wd, allowed_tags=re.compile('(NN|JJ|RB)'))  # lemmatize
            if lemmatized_word:
                doc_out = doc_out + [lemmatized_word[0].split(b'/')[0].decode('utf-8')]
        else:
            continue
    data_processed.append(doc_out)

# Print a small sample
print(data_processed[0][:5])
#> ['anarchism', 'originated', 'term', 'abuse', 'first']
```

The `data_processed` is now processed as a list of list of words. You can now use this to create the `Dictionary` and `Corpus`, which will then be used as inputs to the LDA model.

```
 # Step 3: Create the Inputs of LDA model: Dictionary and Corpus
dct = corpora.Dictionary(data_processed)
corpus = [dct.doc2bow(line) for line in data_processed]
```

We have the Dictionary and Corpus created. Let's build a LDA topic model with 7 topics, using LdaMulticore() . 7 topics is an arbitrary choice for now.

```
 # Step 4: Train the LDA model
lda_model = LdaMulticore(corpus=corpus,
                         id2word=dct,
                         random_state=100,
                         num_topics=7,
                         passes=10,
                         chunksize=1000,
                         batch=False,
                         alpha='asymmetric',
                         decay=0.5,
                         offset=64,
                         eta=None,
                         eval_every=0,
                         iterations=100,
                         gamma_threshold=0.001,
                         per_word_topics=True)

# save the model
lda_model.save('lda_model.model')

# See the topics
lda_model.print_topics(-1)
# [(0, '0.001*"also" + 0.000*"first" + 0.000*"state" + 0.000*"american" + 0.000*"time" + 0.000*
#  (1, '0.001*"also" + 0.001*"state" + 0.001*"ammonia" + 0.000*"first" + 0.000*"many" + 0.000*"
#  (2, '0.005*"also" + 0.004*"american" + 0.004*"state" + 0.004*"first" + 0.003*"year" + 0.003*
#  (3, '0.001*"atheism" + 0.001*"also" + 0.001*"first" + 0.001*"atheist" + 0.001*"american" + 0
#  (4, '0.001*"state" + 0.001*"also" + 0.001*"many" + 0.000*"world" + 0.000*"agave" + 0.000*"ti
#  (5, '0.001*"also" + 0.001*"abortion" + 0.001*"first" + 0.001*"american" + 0.000*"state" + 0.
#  (6, '0.005*"also" + 0.004*"first" + 0.003*"time" + 0.003*"many" + 0.003*"state" + 0.003*"wor
```

The lda_model.print_topics shows what words contributed to which of the 7 topics, along with the weightage of the word's contribution to that topic.

You can see the words like 'also', 'many' coming across different topics. So I would add such words to the stop_words list to remove them and further tune to topic model for optimal number of topics (https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/).

LdaMulticore() supports parallel processing. Alternately you could also try and see what topics the LdaModel() gives.

12. How to interpret the LDA Topic Model's output?

The lda_model object supports indexing. That is, if you pass a document (list of words) to the lda_model , it provides 3 things:

1. The topic(s) that document belongs to along with percentage.
2. The topic(s) each word in that document belongs to.
3. The topic(s) each word in that document belongs to AND the phi values.

So, what is phi value?

Phi value is the probability of the word belonging to that particular topic. And the sum of phi values for a given word adds up to the number of times that word occurred in that document.

For example, in below output for the 0th document, the word with id=0 belongs to topic number 6 and the phi value is 3.999. That means, the word with id=0 appeared 4 times in the 0th document.

```
# Reference: https://github.com/RaRe-Technologies/gensim/blob/develop/docs/note
for c in lda_model[corpus[5:8]]:
    print("Document Topics      : ", c[0])      # [(Topics, Perc Contrib)]
    print("Word id, Topics      : ", c[1][:3])  # [(Word id, [Topics])]
    print("Phi Values (word id) : ", c[2][:2])  # [(Word id, [(Topic, Phi Value)])]
    print("Word, Topics         : ", [(dct[wd], topic) for wd, topic in c[1][:2]])   # [(Word,
    print("Phi Values (word)    : ", [(dct[wd], topic) for wd, topic in c[2][:2]])  # [(Word, [
    print("-----------------------------------------------------\n")

#> Document Topics      :  [(2, 0.96124125), (6, 0.038569752)]
#> Word id, Topics      :  [(0, [2, 6]), (7, [2, 6]), (10, [2, 6])]
#> Phi Values (word id) :  [(0, [(2, 2.887749), (6, 0.112249866)]), (7, [(2, 0.90105206), (6, 0
#> Word, Topics         :  [('ability', [2, 6]), ('absurdity', [2, 6])]
#> Phi Values (word)    :  [('ability', [(2, 2.887749), (6, 0.112249866)]), ('absurdity', [(2,
#> -----------------------------------------------------

#> Document Topics      :  [(6, 0.9997751)]
#> Word id, Topics      :  [(0, [6]), (10, [6]), (16, [6])]
#> Phi Values (word id) :  [(0, [(6, 5.9999967)]), (10, [(6, 2.9999983)])]
#> Word, Topics         :  [('ability', [6]), ('academic', [6])]
#> Phi Values (word)    :  [('ability', [(6, 5.9999967)]), ('academic', [(6, 2.9999983)])]
#> -----------------------------------------------------

#> Document Topics      :  [(6, 0.9998023)]
#> Word id, Topics      :  [(1, [6]), (10, [6]), (15, [6])]
#> Phi Values (word id) :  [(1, [(6, 0.99999917)]), (10, [(6, 5.999997)])]
#> Word, Topics         :  [('able', [6]), ('academic', [6])]
#> Phi Values (word)    :  [('able', [(6, 0.99999917)]), ('academic', [(6, 5.999997)])]
#> -----------------------------------------------------
```

13. How to create a LSI topic model using gensim?

The syntax for using an LSI model is similar to how we built the LDA model, except that we will use the `LsiModel()`.

```
from gensim.models import LsiModel

# Build the LSI Model
lsi_model = LsiModel(corpus=corpus, id2word=dct, num_topics=7, decay=0.5)

# View Topics
pprint(lsi_model.print_topics(-1))
#> [(0, '0.262*"also" + 0.197*"state" + 0.197*"american" + 0.178*"first" + '
#>   '0.151*"many" + 0.149*"time" + 0.147*"year" + 0.130*"person" + 0.130*"world" '
#>   '+ 0.124*"war"'),
#>  (1, '0.937*"agave" + 0.164*"asia" + 0.100*"aruba" + 0.063*"plant" + 0.053*"var" '
#>   '+ 0.052*"state" + 0.045*"east" + 0.044*"congress" + -0.042*"first" + '
#>   '0.041*"maguey"'),
#>  (2, '0.507*"american" + 0.180*"football" + 0.179*"player" + 0.168*"war" + '
#>   '0.150*"british" + -0.140*"also" + 0.114*"ball" + 0.110*"day" + '
#>   '-0.107*"atheism" + -0.106*"god"'),
#>  (3, '-0.362*"apollo" + 0.248*"lincoln" + 0.211*"state" + -0.172*"player" + '
#>   '-0.151*"football" + 0.127*"union" + -0.125*"ball" + 0.124*"government" + '
#>   '-0.116*"moon" + 0.116*"jews"'),
#>  (4, '-0.363*"atheism" + -0.334*"god" + -0.329*"lincoln" + -0.230*"apollo" + '
#>   '-0.215*"atheist" + -0.143*"abraham" + 0.136*"island" + -0.132*"aristotle" + '
#>   '0.124*"aluminium" + -0.119*"belief"'),
#>  (5, '-0.360*"apollo" + 0.344*"atheism" + -0.326*"lincoln" + 0.226*"god" + '
#>   '0.205*"atheist" + 0.139*"american" + -0.130*"lunar" + 0.128*"football" + '
#>   '-0.125*"moon" + 0.114*"belief"'),
#>  (6, '-0.313*"lincoln" + 0.226*"apollo" + -0.166*"football" + -0.163*"war" + '
#>   '0.162*"god" + 0.153*"australia" + -0.148*"play" + -0.146*"ball" + '
#>   '0.122*"atheism" + -0.122*"line"')]
```

14. How to train Word2Vec model using gensim?

A word embedding model is a model that can provide numerical vectors for a given word. Using the Gensim's downloader API, you can download pre-built word embedding models like word2vec, fasttext, GloVe and ConceptNet. These are built on large corpuses of commonly occurring text data such as wikipedia, google news etc.

However, if you are working in a specialized niche such as technical documents, you may not able to get word embeddings for all the words. So, in such cases its desirable to train your own model.

Gensim's `Word2Vec` implementation let's you train your own word embedding model for a given corpus.

```
from gensim.models.word2vec import Word2Vec
from multiprocessing import cpu_count
import gensim.downloader as api

# Download dataset
dataset = api.load("text8")
data = [d for d in dataset]

# Split the data into 2 parts. Part 2 will be used later to update the model
data_part1 = data[:1000]
data_part2 = data[1000:]

# Train Word2Vec model. Defaults result vector size = 100
model = Word2Vec(data_part1, min_count = 0, workers=cpu_count())

# Get the word vector for given word
model['topic']
#> array([ 0.0512,  0.2555,  0.9393, ... ,-0.5669,  0.6737], dtype=float32)

model.most_similar('topic')
#> [('discussion', 0.7590423822402954),
#>  ('consensus', 0.7253159284591675),
#>  ('discussions', 0.7252693176269531),
#>  ('interpretation', 0.7196053266525269),
#>  ('viewpoint', 0.7053568959236145),
#>  ('speculation', 0.7021505832672119),
#>  ('discourse', 0.7001898884773254),
#>  ('opinions', 0.6993060111999512),
#>  ('focus', 0.6959210634231567),
#>  ('scholarly', 0.6884037256240845)]

# Save and Load Model
model.save('newmodel')
model = Word2Vec.load('newmodel')
```

We have trained and saved a Word2Vec model for our document. However, when a new dataset comes, you want to update the model so as to account for new words.

15. How to update an existing Word2Vec model with new data?

On an existing `Word2Vec` model, call the `build_vocab()` on the new datset and then call the `train()` method. `build_vocab()` is called first because the model has to be apprised of what new words to expect in the incoming corpus.

```
# Update the model with new data.
model.build_vocab(data_part2, update=True)
model.train(data_part2, total_examples=model.corpus_count, epochs=model.iter)
model['topic']
# array([-0.6482, -0.5468,  1.0688,  0.82  , ... , -0.8411,  0.3974], dtype=float32)
```

We just saw how to get the word vectors for Word2Vec model we just trained.

16. How to extract word vectors using pre-trained Word2Vec and FastText models?

However, gensim lets you download state of the art pretrained models through the downloader API. Let's see how to extract the word vectors from a couple of these models.

```
import gensim.downloader as api

# Download the models
fasttext_model300 = api.load('fasttext-wiki-news-subwords-300')
word2vec_model300 = api.load('word2vec-google-news-300')
glove_model300 = api.load('glove-wiki-gigaword-300')

# Get word embeddings
word2vec_model300.most_similar('support')
# [('supporting', 0.6251285076141357),
#  ...
#  ('backing', 0.6007589101791382),
#  ('supports', 0.5269277691841125),
#  ('assistance', 0.520713746547699),
#  ('supportive', 0.5110025405883789)]
```

We have 3 different embedding models. You can evaluate which one performs better using the respective model's `evaluate_word_analogies()` on a standard analogies dataset (https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/datasets/questions-words.txt).

```
# Word2ec_accuracy
word2vec_model300.evaluate_word_analogies(analogies="questions-words.txt")[0]
#> 0.7401448525607863

# fasttext_accuracy
fasttext_model300.evaluate_word_analogies(analogies="questions-words.txt")[0]
#> 0.8827876424099353

# GloVe accuracy
glove_model300.evaluate_word_analogies(analogies="questions-words.txt")[0]
#> 0.7195422354510931
```

17. How to create document vectors using Doc2Vec?

Unlike `Word2Vec`, a `Doc2Vec` model provides a vectorised representation of a group of words taken collectively as a single unit. It is not a simple average of the word vectors of the words in the sentence.

Let's use the `text8` dataset to train the `Doc2Vec`.

```
import gensim
import gensim.downloader as api

# Download dataset
dataset = api.load("text8")
data = [d for d in dataset]
```

The training data for Doc2Vec should be a list of TaggedDocument s. To create one, we pass a list of words and a unique integer as input to the models.doc2vec.TaggedDocument() .

```
# Create the tagged document needed for Doc2Vec
def create_tagged_document(list_of_list_of_words):
    for i, list_of_words in enumerate(list_of_list_of_words):
        yield gensim.models.doc2vec.TaggedDocument(list_of_words, [i])

train_data = list(create_tagged_document(data))

print(train_data[:1])
#> [TaggedDocument(words=['anarchism', 'originated', ... 'social', 'or', 'emotional'], tags=[0]
```

The input is prepared. To train the model, you need to initialize the Doc2Vec model, build the vocabulary and then finally train the model.

```
# Init the Doc2Vec model
model = gensim.models.doc2vec.Doc2Vec(vector_size=50, min_count=2, epochs=40)

# Build the Volabulary
model.build_vocab(train_data)

# Train the Doc2Vec model
model.train(train_data, total_examples=model.corpus_count, epochs=model.epochs)
```

To get the document vector of a sentence, pass it as a list of words to the infer_vector() method.

```
print(model.infer_vector(['australian', 'captain', 'elected', 'to', 'bowl']))
#> array([-0.11043505,  0.21719663, -0.21167697, -0.10790558,  0.5607173 ,
#>        ...
#>        0.16428669, -0.31307793, -0.28575218, -0.0113026 ,  0.08981086],
#>       dtype=float32)
```

18. How to compute similarity metrics like cosine similarity and soft cosine similarity?

Soft cosine similarity is similar to cosine similarity (https://en.wikipedia.org/wiki/Cosine_similarity) but in addition considers the semantic relationship between the words through its vector representation.

To compute soft cosines, you will need a word embedding model like Word2Vec or FastText. First, compute the `similarity_matrix`. Then convert the input sentences to bag-of-words corpus and pass them to the `softcossim()` along with the similarity matrix.

```
 from gensim.matutils import softcossim
from gensim import corpora

sent_1 = 'Sachin is a cricket player and a opening batsman'.split()
sent_2 = 'Dhoni is a cricket player too He is a batsman and keeper'.split()
sent_3 = 'Anand is a chess player'.split()

# Prepare the similarity matrix
similarity_matrix = fasttext_model300.similarity_matrix(dictionary, tfidf=None, threshold=0.0,

# Prepare a dictionary and a corpus.
documents = [sent_1, sent_2, sent_3]
dictionary = corpora.Dictionary(documents)

# Convert the sentences into bag-of-words vectors.
sent_1 = dictionary.doc2bow(sent_1)
sent_2 = dictionary.doc2bow(sent_2)
sent_3 = dictionary.doc2bow(sent_3)

# Compute soft cosine similarity
print(softcossim(sent_1, sent_2, similarity_matrix))
#> 0.7868705819999783

print(softcossim(sent_1, sent_3, similarity_matrix))
#> 0.6036445529268666

print(softcossim(sent_2, sent_3, similarity_matrix))
#> 0.60965453519611
```

Below are some useful similarity and distance metrics based on the word embedding models like fasttext and GloVe. We have already downloaded these models using the downloader API.

```
 # Which word from the given list doesn't go with the others?
print(fasttext_model300.doesnt_match(['india', 'australia', 'pakistan', 'china', 'beetroot']))
#> beetroot

# Compute cosine distance between two words.
print(fasttext_model300.distance('king', 'queen'))
#> 0.22957539558410645



# Compute cosine distances from given word or vector to all words in `other_words`.
print(fasttext_model300.distances('king', ['queen', 'man', 'woman']))
#> [0.22957546 0.465837   0.547001  ]



# Compute cosine similarities
print(fasttext_model300.cosine_similarities(fasttext_model300['king'],
                                            vectors_all=(fasttext_model300['queen'],
                                                         fasttext_model300['man'],
                                                         fasttext_model300['woman'],
                                                         fasttext_model300['queen'] + fasttext_m
#> array([0.77042454, 0.534163  , 0.45299897, 0.76572555], dtype=float32)
# Note: Queen + Man is very similar to King.

# Get the words closer to w1 than w2
print(glove_model300.words_closer_than(w1='king', w2='kingdom'))
#> ['prince', 'queen', 'monarch']



# Find the top-N most similar words.
print(fasttext_model300.most_similar(positive='king', negative=None, topn=5, restrict_vocab=Non
#> [('queen', 0.63), ('prince', 0.62), ('monarch', 0.59), ('kingdom', 0.58), ('throne', 0.56)]



# Find the top-N most similar words, using the multiplicative combination objective,
print(glove_model300.most_similar_cosmul(positive='king', negative=None, topn=5))
#> [('queen', 0.82), ('prince', 0.81), ('monarch', 0.79), ('kingdom', 0.79), ('throne', 0.78)]
```

19. How to summarize text documents?

Gensim implements the textrank summarization using the `summarize()` function in the `summarization` module. All you need to do is to pass in the tet string along with either the output summarization `ratio` or the maximum `count` of words in the summarized output.

There is no need to split the sentence into a tokenized list because gensim does the splitting using the built-in `split_sentences()` method in the `gensim.summarization.texcleaner` module.

Let's summarize the clipping from a new article in sample.txt.

```
from gensim.summarization import summarize, keywords
from pprint import pprint

text = " ".join((line for line in smart_open('sample.txt', encoding='utf-8')))

# Summarize the paragraph
pprint(summarize(text, word_count=20))
#> ('the PLA Rocket Force national defense science and technology experts panel, '
#>  'according to a report published by the')

# Important keywords from the paragraph
print(keywords(text))
#> force zhang technology experts pla rocket
```

For more information on summarization with gensim, refer to this tutorial (https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/summarization_tutorial.ipynb).

20. Conclusion

We have covered a lot of ground about the various features of gensim and get a good grasp on how to work with and manipulate texts. The above examples should serve as nice templates to get you started and build upon for various NLP tasks. Hope you will find it helpful and feel comfortable to use gensim more often in your NLP projects.