Note

Click here to download the full example code

Doc2Vec Model

Introduces Gensim's Doc2Vec model and demonstrates its use on the Lee Corpus.

```
import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %
(message)s', level=logging.INFO)
```

Doc2Vec is a Model that represents each Document as a Vector. This tutorial introduces the model and demonstrates how to train and assess it.

Here's a list of what we'll be doing:

0. Review the relevant models: bag-of-words, Word2Vec, Doc2Vec
1. Load and preprocess the training and test corpora (see Corpus)
2. Train a Doc2Vec Model model using the training corpus
3. Demonstrate how the trained model can be used to infer a Vector
4. Assess the model
5. Test the model on the test corpus

Review: Bag-of-words

Note

Feel free to skip these review sections if you're already familiar with the models.

You may be familiar with the bag-of-words model from the Vector section. This model transforms each document to a fixed-length vector of integers. For example, given the sentences:

- `John likes to watch movies. Mary likes movies too.`
- `John also likes to watch football games. Mary hates football.`

The model outputs the vectors:

- `[1, 2, 1, 1, 2, 1, 1, 0, 0, 0, 0]`
- `[1, 1, 1, 1, 0, 1, 0, 1, 2, 1, 1]`

Each vector has 10 elements, where each element counts the number of times a particular word occurred in the document. The order of elements is arbitrary. In the example above, the order of the elements corresponds to the words: `["John", "likes", "to", "watch", "movies", "Mary", "too", "also", "football", "games", "hates"]`.

Bag-of-words models are surprisingly effective, but have several weaknesses.

First, they lose all information about word order: “John likes Mary” and “Mary likes John” correspond to identical vectors. There is a solution: bag of n-grams models consider word phrases of length n to represent documents as fixed-length vectors to capture local word order but suffer from data sparsity and high dimensionality.

Second, the model does not attempt to learn the meaning of the underlying words, and as a consequence, the distance between vectors doesn’t always reflect the difference in meaning. The `Word2Vec` model addresses this second problem.

Review: `Word2Vec` Model

`Word2Vec` is a more recent model that embeds words in a lower-dimensional vector space using a shallow neural network. The result is a set of word-vectors where vectors close together in vector space have similar meanings based on context, and word-vectors distant to each other have differing meanings. For example, `strong` and `powerful` would be close together and `strong` and `Paris` would be relatively far.

Gensim’s **`Word2Vec`** class implements this model.

With the `Word2Vec` model, we can calculate the vectors for each **word** in a document. But what if we want to calculate a vector for the **entire document**? We could average the vectors for each word in the document – while this is quick and crude, it can often be useful. However, there is a better way...

Introducing: Paragraph Vector

Important

In Gensim, we refer to the Paragraph Vector model as `Doc2Vec`.

Le and Mikolov in 2014 introduced the Doc2Vec algorithm, which usually outperforms such simple-averaging of `Word2Vec` vectors.

The basic idea is: act as if a document has another floating word-like vector, which contributes to all training predictions, and is updated like other word-vectors, but we will call it a doc-vector. Gensim's `Doc2Vec` class implements this algorithm.

There are two implementations:

1. Paragraph Vector - Distributed Memory (PV-DM)
2. Paragraph Vector - Distributed Bag of Words (PV-DBOW)

Important

Don't let the implementation details below scare you. They're advanced material: if it's too much, then move on to the next section.

PV-DM is analogous to Word2Vec CBOW. The doc-vectors are obtained by training a neural network on the synthetic task of predicting a center word based an average of both context word-vectors and the full document's doc-vector.

PV-DBOW is analogous to Word2Vec SG. The doc-vectors are obtained by training a neural network on the synthetic task of predicting a target word just from the full document's doc-vector. (It is also common to combine this with skip-gram testing, using both the doc-vector and nearby word-vectors to predict a single target word, but only one at a time.)

Prepare the Training and Test Data

For this tutorial, we'll be training our model using the Lee Background Corpus included in gensim. This corpus contains 314 documents selected from the Australian Broadcasting Corporation's news mail service, which provides text e-mails of headline stories and covers a number of broad topics.

And we'll test our model by eye using the much shorter Lee Corpus which contains 50 documents.

```
import os
import gensim
# Set file names for train and test data
test_data_dir = os.path.join(gensim.__path__[0], 'test',
'test_data')
lee_train_file = os.path.join(test_data_dir,
'lee_background.cor')
lee_test_file = os.path.join(test_data_dir, 'lee.cor')
```

Define a Function to Read and Preprocess Text

Below, we define a function to:

- open the train/test file (with latin encoding)
- read the file line-by-line
- pre-process each line (tokenize text into individual words, remove punctuation, set to lowercase, etc)

The file we're reading is a **corpus**. Each line of the file is a **document**.

Important

To train the model, we'll need to associate a tag/number with each document of the training corpus. In our case, the tag is simply the zero-based line number.

```
import smart_open

def read_corpus(fname, tokens_only=False):
    with smart_open.open(fname, encoding="iso-8859-1") as f:
        for i, line in enumerate(f):
            tokens = gensim.utils.simple_preprocess(line)
            if tokens_only:
                yield tokens
            else:
                # For training data, add tags
                yield
gensim.models.doc2vec.TaggedDocument(tokens, [i])

train_corpus = list(read_corpus(lee_train_file))
test_corpus = list(read_corpus(lee_test_file,
tokens_only=True))
```

Let's take a look at the training corpus

```
print(train_corpus[:2])
```

Out:

```
[TaggedDocument(words=['hundreds', 'of', 'people', 'have',
'been', 'forced', 'to', 'vacate', 'their', 'homes', 'in',
'the', 'southern', 'highlands', 'of', 'new', 'south',
'wales', 'as', 'strong', 'winds', 'today', 'pushed', 'huge',
'bushfire', 'towards', 'the', 'town', 'of', 'hill', 'top',
'new', 'blaze', 'near', 'goulburn', 'south', 'west', 'of',
'sydney', 'has', 'forced', 'the', 'closure', 'of', 'the',
'hume', 'highway', 'at', 'about', 'pm', 'aedt', 'marked',
'deterioration', 'in', 'the', 'weather', 'as', 'storm',
'cell', 'moved', 'east', 'across', 'the', 'blue',
'mountains', 'forced', 'authorities', 'to', 'make',
'decision', 'to', 'evacuate', 'people', 'from', 'homes',
'in', 'outlying', 'streets', 'at', 'hill', 'top', 'in',
'the', 'new', 'south', 'wales', 'southern', 'highlands',
'an', 'estimated', 'residents', 'have', 'left', 'their',
'homes', 'for', 'nearby', 'mittagong', 'the', 'new', 'south',
'wales', 'rural', 'fire', 'service', 'says', 'the',
'weather', 'conditions', 'which', 'caused', 'the', 'fire',
'to', 'burn', 'in', 'finger', 'formation', 'have', 'now',
'eased', 'and', 'about', 'fire', 'units', 'in', 'and',
'around', 'hill', 'top', 'are', 'optimistic', 'of',
```

And the testing corpus looks like this:

```
print(test_corpus[:2])
```

Out:

```
[['the', 'national', 'executive', 'of', 'the', 'strife',
'torn', 'democrats', 'last', 'night', 'appointed', 'little',
'known', 'west', 'australian', 'senator', 'brian', 'greig',
'as', 'interim', 'leader', 'shock', 'move', 'likely', 'to',
'provoke', 'further', 'conflict', 'between', 'the', 'party',
'senators', 'and', 'its', 'organisation', 'in', 'move', 'to',
'reassert', 'control', 'over', 'the', 'party', 'seven',
'senators', 'the', 'national', 'executive', 'last', 'night',
'rejected', 'aden', 'ridgeway', 'bid', 'to', 'become',
'interim', 'leader', 'in', 'favour', 'of', 'senator',
'greig', 'supporter', 'of', 'deposed', 'leader', 'natasha',
'stott', 'despoja', 'and', 'an', 'outspoken', 'gay',
'rights', 'activist'], ['cash', 'strapped', 'financial',
'services', 'group', 'amp', 'has', 'shelved', 'million',
'plan', 'to', 'buy', 'shares', 'back', 'from', 'investors',
'and', 'will', 'raise', 'million', 'in', 'fresh', 'capital',
'after', 'profits', 'crashed', 'in', 'the', 'six', 'months',
'to', 'june', 'chief', 'executive', 'paul', 'batchelor',
'said', 'the', 'result', 'was', 'solid', 'in', 'what', 'he',
'described', 'as', 'the', 'worst', 'conditions', 'for',
'stock', 'markets', 'in', 'years', 'amp', 'half', 'year',
```

Notice that the testing corpus is just a list of lists and does not contain any tags.

Training the Model

Now, we'll instantiate a Doc2Vec model with a vector size with 50 dimensions and iterating over the training corpus 40 times. We set the minimum word count to 2 in order to discard words with very few occurrences. (Without a variety of representative examples, retaining such infrequent words can often make a model worse!) Typical iteration counts in the published Paragraph Vector paper results, using 10s-of-thousands to millions of docs, are 10-20. More iterations take more time and eventually reach a point of diminishing returns.

However, this is a very very small dataset (300 documents) with shortish documents (a few hundred words). Adding training passes can sometimes help with such small datasets.

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

Build a vocabulary

```
model.build_vocab(train_corpus)
```

Out:

```
2020-09-30 21:08:55,026 : INFO : collecting all words and
their counts
2020-09-30 21:08:55,027 : INFO : PROGRESS: at example #0,
processed 0 words (0/s), 0 word types, 0 tags
2020-09-30 21:08:55,043 : INFO : collected 6981 word types
and 300 unique tags from a corpus of 300 examples and 58152
words
2020-09-30 21:08:55,043 : INFO : Loading a fresh vocabulary
2020-09-30 21:08:55,064 : INFO : effective_min_count=2
retains 3955 unique words (56% of original 6981, drops 3026)
2020-09-30 21:08:55,064 : INFO : effective_min_count=2 leaves
55126 word corpus (94% of original 58152, drops 3026)
2020-09-30 21:08:55,098 : INFO : deleting the raw counts
dictionary of 6981 items
2020-09-30 21:08:55,100 : INFO : sample=0.001 downsamples 46
most-common words
2020-09-30 21:08:55,100 : INFO : downsampling leaves
estimated 42390 word corpus (76.9% of prior 55126)
2020-09-30 21:08:55,149 : INFO : estimated required memory
for 3955 words and 50 dimensions: 3679500 bytes
2020-09-30 21:08:55,149 : INFO : resetting layer weights
```

Essentially, the vocabulary is a list (accessible via `model.wv.index_to_key`) of all of the unique words extracted from the training corpus. Additional attributes for each word are available using the `model.wv.get_vecattr()` method, For example, to see how many times `penalty` appeared in the training corpus:

```
print(f"Word 'penalty' appeared
{model.wv.get_vecattr('penalty', 'count')} times in the
training corpus.")
```

Out:

```
Word 'penalty' appeared 4 times in the training corpus.
```

Next, train the model on the corpus. If optimized Gensim (with BLAS library) is being used, this should take no more than 3 seconds. If the BLAS library is not being used, this should take no more than 2 minutes, so use optimized Gensim with BLAS if you value your time.

```
model.train(train_corpus, total_examples=model.corpus_count,
epochs=model.epochs)
```

Out:

```
2020-09-30 21:08:56,478 : INFO : worker thread finished;
awaiting finish of 1 more threads
2020-09-30 21:08:56,479 : INFO : worker thread finished;
awaiting finish of 0 more threads
2020-09-30 21:08:56,479 : INFO : EPOCH - 16 : training on
58152 raw words (42799 effective words) took 0.1s, 829690
effective words/s
2020-09-30 21:08:56,530 : INFO : worker thread finished;
awaiting finish of 2 more threads
2020-09-30 21:08:56,530 : INFO : worker thread finished;
awaiting finish of 1 more threads
2020-09-30 21:08:56,533 : INFO : worker thread finished;
awaiting finish of 0 more threads
2020-09-30 21:08:56,534 : INFO : EPOCH - 17 : training on
58152 raw words (42733 effective words) took 0.1s, 794744
effective words/s
2020-09-30 21:08:56,583 : INFO : worker thread finished;
awaiting finish of 2 more threads
2020-09-30 21:08:56,585 : INFO : worker thread finished;
awaiting finish of 1 more threads
2020-09-30 21:08:56,587 : INFO : worker thread finished;
awaiting finish of 0 more threads
```

Now, we can use the trained model to infer a vector for any piece of text by passing a list of words to the `model.infer_vector` function. This vector can then be compared with other vectors via cosine similarity.

```
vector = model.infer_vector(['only', 'you', 'can', 'prevent',
'forest', 'fires'])
print(vector)
```

Out:

```
[-0.08478509  0.05011684  0.0675064  -0.19926868 -0.1235586
0.01768214
 -0.12645927  0.01062329  0.06113973  0.35424358  0.01320948
0.07561274
 -0.01645093  0.0692549   0.08346193 -0.01599065  0.08287009
-0.0139379
 -0.17772709 -0.26271465  0.0442089  -0.04659882 -0.12873884
0.28799203
 -0.13040264  0.12478471 -0.14091878 -0.09698066 -0.07903259
-0.10124907
 -0.28239366  0.13270256  0.04445919 -0.24210942 -0.1907376
-0.07264525
 -0.14167067 -0.22816683 -0.00663796  0.23165748 -0.10436232
-0.01028251
 -0.04064698  0.08813146  0.01072008 -0.149789    0.05923386
0.16301566
  0.05815683  0.1258063 ]
```

Note that `infer_vector()` does *not* take a string, but rather a list of string tokens, which should have already been tokenized the same way as the `words` property of original training document objects.

Also note that because the underlying training/inference algorithms are an iterative approximation problem that makes use of internal randomization, repeated inferences of the same text will return slightly different vectors.

Assessing the Model

To assess our new model, we'll first infer new vectors for each document of the training corpus, compare the inferred vectors with the training corpus, and then returning the rank of the document based on self-similarity. Basically, we're pretending as if the training corpus is some new unseen data and then seeing how they compare with the trained model. The expectation is that we've likely overfit our model (i.e., all of the ranks will be less than 2) and so we should be able to find similar documents very easily. Additionally, we'll keep track of the second ranks for a comparison of less similar documents.

```
ranks = []
second_ranks = []
for doc_id in range(len(train_corpus)):
    inferred_vector =
model.infer_vector(train_corpus[doc_id].words)
    sims = model.dv.most_similar([inferred_vector],
topn=len(model.dv))
    rank = [docid for docid, sim in sims].index(doc_id)
    ranks.append(rank)

    second_ranks.append(sims[1])
```

Let's count how each document ranks with respect to the training corpus

NB. Results vary between runs due to random seeding and very small corpus

```python
import collections

counter = collections.Counter(ranks)
print(counter)
```

Out:

```
Counter({0: 292, 1: 8})
```

Basically, greater than 95% of the inferred documents are found to be most similar to itself and about 5% of the time it is mistakenly most similar to another document. Checking the inferred-vector against a training-vector is a sort of 'sanity check' as to whether the model is behaving in a usefully consistent manner, though not a real 'accuracy' value.

This is great and not entirely surprising. We can take a look at an example:

```python
print('Document ({}): «{}»\n'.format(doc_id, ' '.join(train_corpus[doc_id].words)))
print(u'SIMILAR/DISSIMILAR DOCS PER MODEL %s:\n' % model)
for label, index in [('MOST', 0), ('SECOND-MOST', 1), ('MEDIAN', len(sims)//2), ('LEAST', len(sims) - 1)]:
    print(u'%s %s: «%s»\n' % (label, sims[index], ' '.join(train_corpus[sims[index][0]].words)))
```

Out:

```
Document (299): «australia will take on france in the doubles
rubber of the davis cup tennis final today with the tie
levelled at wayne arthurs and todd woodbridge are scheduled
to lead australia in the doubles against cedric pioline and
fabrice santoro however changes can be made to the line up up
to an hour before the match and australian team captain john
fitzgerald suggested he might do just that we ll make team
appraisal of the whole situation go over the pros and cons
and make decision french team captain guy forget says he will
not make changes but does not know what to expect from
australia todd is the best doubles player in the world right
now so expect him to play he said would probably use wayne
arthurs but don know what to expect really pat rafter
salvaged australia davis cup campaign yesterday with win in
the second singles match rafter overcame an arm injury to
defeat french number one sebastien grosjean in three sets the
australian says he is happy with his form it not very pretty
tennis there isn too many consistent bounces you are playing
like said bit of classic old grass court rafter said rafter
levelled the score after lleyton hewitt shock five set loss
to nicholas escude in the first singles rubber but rafter
```

Notice above that the most similar document (usually the same text) is has a similarity score approaching 1.0. However, the similarity score for the second-ranked documents should be significantly lower (assuming the documents are in fact different) and the reasoning becomes obvious when we examine the text itself.

We can run the next cell repeatedly to see a sampling other target-document comparisons.

```
# Pick a random document from the corpus and infer a vector
from the model
import random
doc_id = random.randint(0, len(train_corpus) - 1)

# Compare and print the second-most-similar document
print('Train Document ({}): «{}»\n'.format(doc_id, '
'.join(train_corpus[doc_id].words)))
sim_id = second_ranks[doc_id]
print('Similar Document {}: «{}»\n'.format(sim_id, '
'.join(train_corpus[sim_id[0]].words)))
```

Out:

```
Train Document (292): «rival afghan factions are deadlocked
over the shape of future government the northern alliance has
demanded day adjournment of power sharing talks in germany
after its president burhanuddin rabbani objected to the
appointment system for an interim administration president
rabbani has objected to the plans for an interim government
to be drawn up by appointment as discussed in bonn saying the
interim leaders should be voted in by afghans themselves he
also says there is no real need for sizeable international
security force president rabbani says he would prefer local
afghan factions drew up their own internal security forces of
around personnel but if the world insisted there should be an
international security presence there should be no more than
or personnel in their security forces he says president
rabbani objections are likely to cast doubt on his delegation
ability to commit the northern alliance to any course of
action decided upon in bonn he now threatens to undermine the
very process he claims to support in the quest for stable
government in afghanistan»

Similar Document (13, 0.7867921590805054): «talks between
```

Testing the Model

Using the same approach above, we'll infer the vector for a randomly chosen test document, and compare the document to our model by eye.

```
# Pick a random document from the test corpus and infer a
vector from the model
doc_id = random.randint(0, len(test_corpus) - 1)
inferred_vector = model.infer_vector(test_corpus[doc_id])
sims = model.dv.most_similar([inferred_vector],
topn=len(model.dv))

# Compare and print the most/median/least similar documents
from the train corpus
print('Test Document ({}): «{}»\n'.format(doc_id, '
'.join(test_corpus[doc_id])))
print(u'SIMILAR/DISSIMILAR DOCS PER MODEL %s:\n' % model)
for label, index in [('MOST', 0), ('MEDIAN', len(sims)//2),
('LEAST', len(sims) - 1)]:
    print(u'%s %s: «%s»\n' % (label, sims[index], '
'.join(train_corpus[sims[index][0]].words)))
```

Out:

```
Test Document (49): «labor needed to distinguish itself from
the government on the issue of asylum seekers greens leader
bob brown has said his senate colleague kerry nettle intends
to move motion today on the first anniversary of the tampa
crisis condemning the government over its refugee policy and
calling for an end to mandatory detention we greens want to
bring the government to book over its serial breach of
international obligations as far as asylum seekers in this
country are concerned senator brown said today»

SIMILAR/DISSIMILAR DOCS PER MODEL
Doc2Vec(dm/m,d50,n5,w5,mc2,s0.001,t3):

MOST (218, 0.8016394376754761): «refugee support groups are
strongly critical of federal government claims that the
pacific solution program is working well the immigration
minister philip ruddock says he is pleased with the program
which uses pacific island nations to process asylum seekers
wanting to come to australia president of the hazara ethnic
society of australia hassan ghulam says the australian
government is bullying smaller nations into accepting asylum
```

Conclusion

Let’s review what we’ve seen in this tutorial:

0. Review the relevant models: bag-of-words, Word2Vec, Doc2Vec
1. Load and preprocess the training and test corpora (see Corpus)
2. Train a Doc2Vec Model model using the training corpus
3. Demonstrate how the trained model can be used to infer a Vector
4. Assess the model
5. Test the model on the test corpus

That’s it! Doc2Vec is a great way to explore relationships between documents.

Additional Resources

If you'd like to know more about the subject matter of this tutorial, check out the links below.

- Word2Vec Paper
- Doc2Vec Paper
- Dr. Michael D. Lee's Website
- Lee Corpus
- IMDB Doc2Vec Tutorial

Total running time of the script: (0 minutes 7.863 seconds)

Estimated memory usage: 37 MB

Download Python source code: run_doc2vec_lee.py

Download Jupyter notebook: run_doc2vec_lee.ipynb