Representation

- vectors, n-grams, and more

Outline

- Introducing Gensim
- Vectors and why we need them
- Vector transformations in Gensim
- n-grams and some more preprocessing
- Summary

Introducing Gensim

- This section will introduce the data structures largely used in text analysis involving machine learning techniques - vectors. we will be discussing converting our textual representations to numerical representations, in particular, moving from strings to vectors.

Introducing Gensim

- We will be exploring different kinds of ways of representing our strings as vectors, such as bag-of-words, TF-IDF (term frequency-inverse document frequency), LSI (latent semantic indexing), and the more recently popular word2vec, and gensim includes methods to do all of the above.

Introducing Gensim

- Gensim included novel implementations of Latent Dirichlet allocation (LDA) and Latent Semantic Analysis among its primary algorithms, as well as TF-IDF and Random projection implementations. It has since grown to be one of the largest NLP/Information Retreival Python libraries, and is both memory-efficient and scalable.

Introducing Gensim

- Gensim manages to be scalable because it uses Python's in-built generators and iterators for streamed data-processing, so the data-set is never actually completely loaded in the RAM. Matrix multiplications are performed by NumPy, which is further built on FORTRAN/C. Since all the heavy lifting is passed on to these low level BLAS libraries, Gensim offers the ease-of-use of Python with the power of C.

Introducing Gensim

- The primary features of Gensim are its memory-independent nature, multicore implementations of latent semantic analysis, latent Dirichlet allocation, random projections, hierarchical Dirichlet process (HDP), and word2vec deep learning, as well as the ability to use LSA and LDA on a cluster of computers. It also seamlessly plugs into the Python scientific computing ecosystem and can be extended with other vector space algorithms.

Vectors and why we need them

- We're now moving toward the machine learning part of text analysis - this means that we will now start playing a little less with words and a little more with numbers. Even when we used spaCy, the POS-tagging and NER-tagging, for example, was done through statistical models - but the inner workings were largely hidden for us.

Vectors and why we need them

- For Gensim however, we're expected to pass vectors as inputs to the IR algorithms (such as LDA or LSI), largely because what's going on under the hood is mathematical operations involving matrices. This means that we have to represent what was previously a string as a vector - and these kind of representations or models are called Vector Space Models.

Vectors and why we need them

- From a mathematical perspective, a vector is a geometric object that has magnitude and direction. We don't need to pay as much attention to this, and rather think of vectors as a way of projecting words onto a mathematical space while preserving the information provided by these words.

Vectors and why we need them

- Machine learning algorithms use these vectors to make predictions. We can understand machine learning as a suite of statistical algorithms and the study of these algorithms. The purpose of these algorithms is to learn from the provided data by decreasing the error of their predictions.

Bag-of-words

- The bag-of-words model is arguably the most straightforward form of representing a sentence as a vector. Let's start with an example:

S1:"The dog sat by the mat."

S2:"The cat loves the dog."

- If we follow the same preprocessing steps of spaCy's Language Models, we will end up with the following sentences:

S1:"dog sat mat."

S2:"cat love dog."

Bag-of-words

- As Python lists, these will now look like this:

S1:['dog', 'sat', 'mat']

S2:['cat', 'love', 'dog']

- If we want to represent this as a vector, we would need to first construct our vocabulary, which would be the unique words found in the sentences. Our vocabulary vector is now as follows:

Vocab = ['dog', 'sat', 'mat', 'love', 'cat']

Bag-of-words

- This means that our representation of our sentences will also be vectors with a length of 5 - we can also say that our vectors will have 5 dimensions. We can also think of mapping of each word in our vocabulary to a number (or index), in which case we can also refer to our vocabulary as a dictionary.
- The bag-of-words model involves using word frequencies to construct our vectors. What will our sentences now look like?

S1:[1, 1, 1, 0, 0]

S2:[1, 0, 0, 1, 1]

Bag-of-words

- It's easy enough to understand - there is 1 occurrence of dog, the first word in the vocabulary, and 0 occurrences of love in the first sentence, so the appropriate indexes are given the value based on the word frequency. If the first sentence has 2 occurrences of the word dog, it would be represented as:

S1: [2, 1, 1, 0, 0]

Bag-of-words

- One important feature of the bag-of-words model which we must remember is that it is an order less document representation - only the counts of the words matter. We can see that in our example above as well, where by looking at the resulting sentence vectors we do not know which words came first. This leads to a loss in spatial information, and by extension, semantic information. However, in a lot of information retrieval algorithms, the order of the words is not important, and just the occurrences of the words are enough for us to start with.

Bag-of-words

- An example where the bag of words model can be used is in spam filtering - emails that are marked as spam are likely to contain spam-related words, such as buy, money, and stock. By converting the text in emails into a bag of words models, we can use Bayesian probability to determine if it is more likely for a mail to be in the spam folder or not.

TF-IDF

- TF-IDF is short for term frequency-inverse document frequency. Largely used in searchengines to find relevant documents based on a query, it is a rather intuitive approach to converting our sentences into vectors.
- TF-IDF tries to encode two different kinds of information - term frequency and inverse document frequency. Term frequency (TF) is the number of times a word appears in a document.

TF-IDF

- IDF helps us understand the importance of a word in a document. By calculating the logarithmically scaled inverse fraction of the documents that contain the word (obtained by dividing the total number of documents by the number of documents containing the term) and then taking the logarithm of that quotient, we can have a measure of how common or rare the word is among all documents.

TF-IDF

- In case the preceding explanation wasn't very clear, expressing them as formulas will help!

TF(t) = (number of times term t appears in a document) / (total number of terms in the document)

IDF(t) = log_e (total number of documents / number of documents with term t in it)

TF-IDF

- TF-IDF is simply the product of these two factors - TF and IDF. Together it encapsulates more information into the vector representation, instead of just using the count of the words like in the bag-of-words vector representation. TF-IDF makes rare words more prominent and ignores common words such as is, of, and that, which may appear a lot of times, but have little importance.

Other representations

- It's possible to extend these representations - indeed, topic models, which we will explore later, are one such example. Word vectors are also an interesting representation of words, where we train a shallow neural network (a neural network with 1 or 2 layers) to describe words as vectors, where each feature is a semantic decoding of the word.

Vector transformations in Gensim

- Now that we know what vector transformations are, let's get used to creating them, and using them. We will be performing these transformations with Gensim, but even scikit-learn can be used. We'll also have a look at scikit-learn's approach later on.

Vector transformations in Gensim

- Let's create our corpus now. We discussed earlier that a corpus is a collection of documents. In our examples, each document would just be one sentence, but this is obviously not the case in most real-world examples we will be dealing with. We should also note that once we are done with preprocessing, we get rid of all punctuation marks - as for as our vector representation is concerned, each document is just one sentence.

Vector transformations in Gensim

```
from gensim import corpora
documents = [u"Football club Arsenal defeat local rivals this weekend.",
u"Weekend football frenzy takes over London.", u"Bank open for takeover
bids after losing millions.", u"London football clubs bid to move to
Wembley stadium.", u"Arsenal bid 50 million pounds for striker Kane.",
u"Financial troubles result in loss of millions for bank.", u"Western bank
files for bankruptcy after financial losses.", u"London football club is
taken over by oil millionaire from Russia.", u"Banking on finances not
working for Russia."]
```

Vector transformations in Gensim

```
import spacy
nlp = spacy.load("en")
texts = []
for document in documents:
  text = []
  doc = nlp(document)
  for w in doc:
    if not w.is_stop and not w.is_punct and not w.like_num:
      text.append(w.lemma_)
  texts.append(text)
print(texts)
```

Vector transformations in Gensim

[[u'football', u'club', u'arsenal', u'defeat', u'local', u'rival',u'weekend'],
[u'weekend', u'football', u'frenzy', u'take', u'london'],
[u'bank', u'open', u'bid', u'lose', u'million'],
[u'london', u'football', u'club', u'bid', u'wembley', u'stadium'],
[u'arsenal', u'bid', u'pound', u'striker', u'kane'],
[u'financial', u'trouble', u'result', u'loss', u'million', u'bank'],
[u'western', u'bank', u'file', u'bankruptcy', u'financial', u'loss'],
[u'london', u'football', u'club', u'take', u'oil', u'millionaire',u'russia'],
[u'bank', u'finance', u'work', u'russia']]

Vector transformations in Gensim

```
dictionary = corpora.Dictionary(texts)
print(dictionary.token2id)
{u'pound': 17, u'financial': 22, u'kane': 18, u'arsenal': 3, u'oil': 27,
u'london': 7, u'result': 23, u'file': 25, u'open': 12, u'bankruptcy': 26,
u'take': 9, u'stadium': 16, u'wembley': 15, u'local': 4, u'defeat': 5,
u'football': 2, u'finance': 31, u'club': 0, u'bid': 10, u'million': 11,
u'striker': 19, u'frenzy': 8, u'western': 24, u'trouble': 21, u'weekend':
6, u'bank': 13, u'loss': 20, u'rival': 1, u'work': 30, u'millionaire': 29,
u'lose': 14, u'russia': 28}
```

Vector transformations in Gensim

- There are 32 unique words in our corpus, all of which are represented in our dictionary with each word being assigned an index value. When we refer to a word's word_id henceforth, it means we are talking about the words integer-id mapping made by the dictionary.
- We will be using the doc2bow method, which, as the name suggests, helps convert our document to bag-of-words.

Vector transformations in Gensim

corpus = [dictionary.doc2bow(text) for text in texts]

[[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)],
[(2, 1), (6, 1), (7, 1), (8, 1), (9, 1)], [(10, 1), (11, 1), (12, 1), (13,1), (14, 1)],
[(0, 1), (2, 1), (7, 1), (10, 1), (15, 1), (16, 1)], [(3, 1), (10, 1), (17,1), (18, 1), (19, 1)],
[(11, 1), (13, 1), (20, 1), (21, 1), (22, 1), (23, 1)],
[(13, 1), (20, 1), (22, 1), (24, 1), (25, 1), (26, 1)],
[(0, 1), (2, 1), (7, 1), (9, 1), (27, 1), (28, 1), (29, 1)], [(13, 1), (28,1), (30, 1), (31, 1)]]

Vector transformations in Gensim

This is a list of lists, where each individual list represents a documents bag-of-words representation.

Unlike the example we demonstrated, where an absence of a word was a 0, we use tuples that represent (word_id, word_count). We can easily verify this by checking the original sentence, mapping each word to its integer ID and reconstructing our list. We can also notice in this case each document has not greater than one count of each word - in smaller corpuses, this tends to happen.

Vector transformations in Gensim

Our corpus is assembled, and we are ready to work machine learning/information retrieval magic on them whenever we would like.

We previously mentioned how Gensim is powerful because it uses streaming corpuses. But in this case, the entire list is loaded into the RAM. This is not a bother for us because it is a toy example, but in any real-world cases, this might cause problems. How do we get past this?

Vector transformations in Gensim

We can start by storing the corpus, once it is created, to disk. One way to do this is as follows:

```
corpora.MmCorpus.serialize('/tmp/example.mm', corpus)
```

By storing the corpus to disk and then later loading from disk, we are being far more memory efficient, because at most one vector resides in the RAM at a time.

Vector transformations in Gensim

Converting a bag of words representation into TF-IDF, for example, is also made very easy with Gensim. We first choose the model/representation we want from the Gensim models' directory.

```
from gensim import models
tfidf = models.TfidfModel(corpus)
```

Vector transformations in Gensim

- This means that tfidf now represents a TF-IDF table trained on our corpus. Note that in case of TFIDF, the training consists simply of going through the supplied corpus once and computing document frequencies of all its features. Training other models, such as latent semantic analysis or latent dirichlet allocation, is much more involved and, consequently, takes much more time.

Vector transformations in Gensim

- We will explore those transformations on the chapters on topic modelling. It is also important to note that all such vector transformations require the same input feature space - which means the same dictionary (and of course, vocabulary).

Vector transformations in Gensim

What does a TF-IDF representation of our corpus look like? All we have to do is this:

```
for document in tfidf[corpus]:
    print(document)
```

Vector transformations in Gensim

[(0, 0.2404682937058529 3), (1, 0.4809365874117058 6), (2, 0.1774993848325405 7), (3, 0.329217986122123 2), (4, 0.4809365874117058 6), (5, 0.4809365874117058 6), (6, 0.329217986122123 2)]

[(2, 0.2421296766697526 6), (6, 0.4490913847888623), (7, 0.3280265464539859 3), (8, 0.6560530929079719), (9, 0.4490913847888623)]

[(10, 0.2959252821810264 3), (11, 0.4051424990000138), (12, 0.5918505643620529), (13, 0.2184344336379748), (14, 0.5918505643620529)]

[(0, 0.2943105474954298 4), (2, 0.2172425325813151 2), (7, 0.2943105474954298 4), (10, 0.2943105474954298 4), (15, 0.5886210949908597), (16, 0.5886210949908597)]

.....

Vector transformations in Gensim

- the float next to each word_id - it is the product of the TF and IDF scores for that particular word, instead of just the word count which was present before. The higher the score, the more important the word in the document.
- We can use this representation as input for our ML algorithms as well, and we can also further chain or link these vector representations by performing another transformation on them.
- Let's move on to a small, but interesting (and useful!) part of text analysis - bi-grams and n_x0002_grams.

n-grams and some more preprocessing

- When working with textual data, context can be very important. As we discussed before, we sometimes lose this context in vector representations, knowing only the count of each word. N-grams, and in particular, bi-grams are going to help us solve this problem, at least to some extent.

n-grams and some more preprocessing

- An n-gram is a contiguous sequence of n items in the text. In our case, we will be dealing with words being the item, but depending on the use case, it could be even letters, syllables, or sometimes in the case of speech, phonemes. A bi-gram is when n = 2.

n-grams and some more preprocessing

- One way bi-grams are calculated in the text is by calculating the conditional probability of a token given by the preceding token. It can also just be calculated by choosing words that appear next to each other, but it is more useful for us to use bi-grams that are more likely to appear as a pair. Such a bi-gram is called a collocation. What this means is that we're trying to find pairs of words that are more likely to appear around each other.

n-grams and some more preprocessing

- For example, New York or Machine Learning could be two possible pairs of words created by bi-grams. In other words, based on the training data, we identify that it is with high probability that the word York follows the word New, and that it is worth considering New York as one identity. We must be careful to get rid of stop words before running a bi-gram model on our corpus, as there could be meaningless bi-grams formed.

n-grams and some more preprocessing

- We can clearly see how this is useful - we can now pick up phrases from our corpus, and New York certainly provides us with more information than the words New and York separately. This means it can be added to our preprocessing pipeline.

n-grams and some more preprocessing

- Gensim approaches bigrams by simply combining the two high probability tokens with an underscore. The tokens new and york will now become new_york instead. Similar to the TF-IDF model, bigrams can be created using another Gensim model - Phrases.

```
import gensim
bigram = gensim.models.Phrases(texts)
```

n-grams and some more preprocessing

- We now have a trained bi-gram model for our corpus. We can perform our transformation on the text the same way we used TF-IDF. We recreate our corpus like this:

```
texts = [bigram[line] for line in texts]
```

n-grams and some more preprocessing

- Each line will now have all possible bi-grams created. It should be noted that in our toy example, we will have no bi-grams or meaningless bi-grams being created. Since by creating new phrases we add words to our dictionary, this step must be done before we create our dictionary. We would have to run this:

```
dictionary = corpora.Dictionary(texts)
corpus = [dictionary.doc2bow(text) for text in texts]
```

n-grams and some more preprocessing

- After we are done creating our bi-grams, we can create tri-grams, and other n-grams by simply running the phrases model multiple times on our corpus. Bi-grams still remains the most used n-gram model, though it is worth one's time to glance over the other uses and kinds of n-gram implementations.

n-grams and some more preprocessing

- This brings us to the end of the preprocessing techniques. It must be noted however that there is no one perfect preprocessing pipeline or set of rules - it depends largely on our use-cases, the kind of data we are working with, and what sort of information we wish to preserve.

n-grams and some more preprocessing

- For example, one popular preprocessing technique involves removing both high frequency and low-frequency words. We can do this in Gensim with the dictionary module. Let's say we would like to get rid of words that occur in less than 20 documents, or in more than 50% of the documents, we would add the following:

```
dictionary.filter_extremes(no_below=20, no_above=0.5)
```

n-grams and some more preprocessing

- We can also remove most frequent tokens or prune out certain token ids. More often than not, it's after multiple iterations of preprocessing and running our algorithms when we figure out the correct preprocessing techniques we wish to use. What is important for us is to know what kind of tools are available to do this, and what is the reason behind doing all of this.

n-grams and some more preprocessing

- We've seen in this section why it makes sense to change our representation of text from words to numbers, and why this is the only language a computer understands. There are different ways computers can interpret words, and TF-IDF and bag of words are two such vector representations. Gensim is a Python package that offers us ways to generate such vector representations, which are later used as inputs into various machine learning and information retrieval algorithms.