NLP For Economists

Text Representation

Sowmya Vajjala

Munich Graduate School of Economics - LMU Munich

Guest Course, October 2020

Goals for this session

- What is text representation and why is it needed?
- What are some common text representation methods?
- Examples of how to apply these methods in Python.

source material: Chapter 3 from `https://practicalnlp.ai`

What is text representation?

- Any form of data text/image/video/audio should be converted into some form of numeric representation so that it can be processed by NLP/Machine Learning algorithms later
- In NLP, this conversion of raw text to a suitable numerical form is called text representation
- We will use text representation and feature extraction/representation as synonyms in this course.

How is text different from any other form of data?

- How do we do feature extraction with images? An image is stored in a computer as a matrix of pixels where each cell[i,j] in the matrix represents pixel i,j of the image. This matrix representation accurately represents the complete image.

How is text different from any other form of data?

- How do we do feature extraction with images? An image is stored in a computer as a matrix of pixels where each cell[i,j] in the matrix represents pixel i,j of the image. This matrix representation accurately represents the complete image.
- A video can be seen as a sequence of frames/images. So, it can be represented as a sequential collection of matrices.

How is text different from any other form of data?

- How do we do feature extraction with images? An image is stored in a computer as a matrix of pixels where each cell[i,j] in the matrix represents pixel i,j of the image. This matrix representation accurately represents the complete image.
- A video can be seen as a sequence of frames/images. So, it can be represented as a sequential collection of matrices.
- Consider speech—it is transmitted as a wave. To represent it mathematically, we sample the wave and record its amplitude (height).

What about text?

Hand crafted features to represent text

- When we know the domain well, it is possible to represent text as a collection of hand crafted features.
- e.g., when I am categorizing sentiment, I can create a list of positive words, and a list of negative words, and use the % of positive and negative words in a document as its two dimensional text representation.
- Assuming we come across a new problem/dataset, and we don't know where to begin, what can we do?

Hand crafted features to represent text

- When we know the domain well, it is possible to represent text as a collection of hand crafted features.
- e.g., when I am categorizing sentiment, I can create a list of positive words, and a list of negative words, and use the % of positive and negative words in a document as its two dimensional text representation.
- Assuming we come across a new problem/dataset, and we don't know where to begin, what can we do?
- in NLP, it is common to use what I will call today as **automatic text representations**, where we don't have to explicitly encode any information.

Consider a Toy corpus

Table 3-1. Our toy corpus

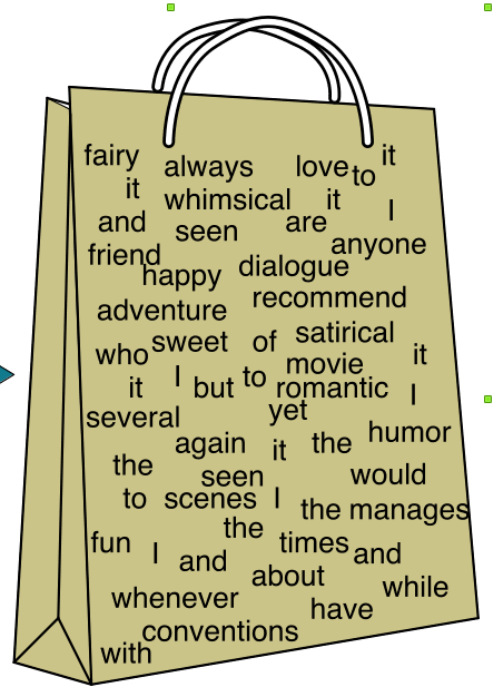
D1	Dog bites man.
D2	Man bites dog.
D3	Dog eats meat.
D4	Man eats food.

Let us start with some simple text representations

- The vocabulary in this corpus, ignoring case-differences and punctuation consists of 6 words: [dog, bites, man, eats, meat, food]
- We can organize the vocabulary in any order. In this example, we simply take the order in which the words appear in the corpus.
- Every document in this corpus can now be represented with a vector of size six.

The Bag of Words Representation

I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!



it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
…	…

source: `https://web.stanford.edu/~jurafsky/slp3/4.pdf`

Bag of Words

- for our toy corpus, where the word IDs are dog = 1, bites = 2, man = 3, meat = 4 , food = 5, eats = 6
- D1 ("Dog bites man") becomes [1 1 1 0 0 0]
- D4 ("Man eats food") becomes [0 0 1 0 1 1].

Code for BoW representation

```
from sklearn.feature_extraction.text import CountVectorizer

documents = ["Dog bites man.", "Man bites dog.", "Dog eats meat.", "Man eats food."]

#remove punctuation and lowercase words:
processed_docs = [doc.lower().replace(".","") for doc in documents]

count_vect = CountVectorizer()

#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#See the BOW rep for first 2 documents
print("BoW representation for 'dog bites man': ", bow_rep[0].toarray())
print("BoW representation for 'man bites dog: ",bow_rep[1].toarray())

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':",

temp.toarray())
```

Bag of Words ...

- If we run this code, we'll notice that the BoW representation for a sentence like "dog and dog are friends" has a value of 2 for the dimension of the word "dog," indicating its frequency in the text.

Bag of Words ...

- If we run this code, we'll notice that the BoW representation for a sentence like "dog and dog are friends" has a value of 2 for the dimension of the word "dog," indicating its frequency in the text.
- Sometimes, we don't care about the frequency of occurrence of words in text and we only want to represent whether a word exists in the text or not. Researchers have shown that such a representation without considering frequency is useful for sentiment analysis.

Bag of Words ...

- If we run this code, we'll notice that the BoW representation for a sentence like "dog and dog are friends" has a value of 2 for the dimension of the word "dog," indicating its frequency in the text.
- Sometimes, we don't care about the frequency of occurrence of words in text and we only want to represent whether a word exists in the text or not. Researchers have shown that such a representation without considering frequency is useful for sentiment analysis.
- In such cases, we just initialize CountVectorizer with the binary=True option keeping everything else the same.

```
count_vect = CountVectorizer(binary=True)
```

- This results in a different representation for the same sentence.

Bag of Words ...

- If we run this code, we'll notice that the BoW representation for a sentence like "dog and dog are friends" has a value of 2 for the dimension of the word "dog," indicating its frequency in the text.
- Sometimes, we don't care about the frequency of occurrence of words in text and we only want to represent whether a word exists in the text or not. Researchers have shown that such a representation without considering frequency is useful for sentiment analysis.
- In such cases, we just initialize CountVectorizer with the binary=True option keeping everything else the same.

```
count_vect = CountVectorizer(binary=True)
```

- This results in a different representation for the same sentence.

Advantages of BoW

- BoW is fairly simple to understand and implement.
- Documents sharing vocabulary have their vector representations closer to each other in Euclidean space. In our toy corpus, distance between D1 and D2 is 0 as compared to the distance between D1 and D4, which is 2.
- We have a fixed-length representation for any sentence of arbitrary length.

Disadvantages of BoW

- The size of the vector increases with the size of the vocabulary. One way to control this is to take the first N most frequent words in the entire corpus.
- t does not capture the similarity between different words that mean the same thing. Say we have three documents: “I run”, “I ran”, and “I ate”. BoW vectors of all three documents will be equally apart.
- This representation does not have any way to handle out of vocabulary words (i.e., new words that were not seen in the corpus that was used to build the vectorizer).
- it is a “bag” of words—word order information is lost in this representation. Both D1 and D2 will have the same representation in this scheme.

Bag of N-grams

- BoN breaking text into chunks of n contiguous words (or tokens) called n-grams, instead of individual words
- This can help us capture some context,
- The corpus vocabulary, V, is then nothing but a collection of all unique n-grams across the text corpus.
- Then, each document in the corpus is represented by a vector of length —V—. This vector simply contains the frequency counts of n-grams present in the document and zero for the n-grams that are not present.

Bag of N-grams with the Toy Corpus

- The set of all bigrams in the corpus is as follows: dog bites, bites man, man bites, bites dog, dog eats, eats meat, man eats, eats food
- The bigram representation for the first two documents is as follows: D1 : [1,1,0,0,0,0,0,0], D2 : [0,0,1,1,0,0,0,0]

Bag of N-grams with the Toy Corpus

- The set of all bigrams in the corpus is as follows: dog bites, bites man, man bites, bites dog, dog eats, eats meat, man eats, eats food
- The bigram representation for the first two documents is as follows: D1 : [1,1,0,0,0,0,0,0], D2 : [0,0,1,1,0,0,0,0]
- Note that the BoW scheme is a special case of the BoN scheme, with n=1. n=2 is called a "bigram model," and n=3 is called a "trigram model." and so on.

Bag of N-grams code

```
#n-gram vectorization example with count vectorizer and
#uni, bi, trigrams
count_vect = CountVectorizer(ngram_range=(1,3))

#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':"
      , temp.toarray())
```

Pros and Cons of Bag of N-grams

- It captures some context and word-order information in the form of n-grams.
- Thus, resulting vector space is able to capture some semantic similarity. Documents having the same n-grams will have their vectors closer to each other in Euclidean space as compared to documents with completely different n-grams.

Pros and Cons of Bag of N-grams

- It captures some context and word-order information in the form of n-grams.
- Thus, resulting vector space is able to capture some semantic similarity. Documents having the same n-grams will have their vectors closer to each other in Euclidean space as compared to documents with completely different n-grams.
- We still have the out of vocabulary issue, and large feature vector issue.

TF-IDF representation

- In BoW/BoNgrams, all the words in the text are treated as equally important—there's no notion of some words in the document being more important than others.
- TF-IDF, or term frequency–inverse document frequency, aims to quantify the importance of a given word relative to other words in the document and in the corpus.

TF-IDF representation

- In BoW/BoNgrams, all the words in the text are treated as equally important—there's no notion of some words in the document being more important than others.
- TF-IDF, or term frequency–inverse document frequency, aims to quantify the importance of a given word relative to other words in the document and in the corpus.

TF(t,d)=(Number of occurrences of term t in document d)/(Total number of terms in the document d) IDF(t)=$\log_e$(Total number of documents in the corpus)/(Number of documents with term t in them)

TFIDF Calculation

Table 3-2. TF-IDF values for our toy corpus

Word	TF score	IDF score	TF-IDF score
dog	⅓ = 0.33	$\log_2(4/3) = 0.4114$	0.4114 * 0.33 = 0.136
bites	⅙ = 0.17	$\log_2(4/2) = 1$	1* 0.17 = 0.17
man	0.33	$\log_2(4/3) = 0.4114$	0.4114 * 0.33 = 0.136
eats	0.17	$\log_2(4/2) = 1$	1* 0.17 = 0.17
meat	1/12 = 0.083	$\log_2(4/1) = 2$	2* 0.083 = 0.17
food	0.083	$\log_2(4/1) = 2$	2* 0.083 = 0.17

The TF-IDF vector representation for a document is then simply the TF-IDF score for each term in that document. So, for D_1 we get

Dog	bites	man	eats	meat	food
0.136	0.17	0.136	0	0	0

TF-IDF code

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()
bow_rep_tfidf = tfidf.fit_transform(processed_docs)
print(tfidf.idf_) #IDF for all words in the vocabulary
print(tfidf.get_feature_names()) #All words in the vocabulary.

temp = tfidf.transform(["dog and man are friends"])
print("Tfidf representation for 'dog and man are friends':\n", temp.toarray())
```

Note: There are some variations of the formula in practice.

Pros and Cons of TFIDF

- We can use the TF-IDF vectors to calculate similarity between two texts using a similarity measure like Euclidean distance or cosine similarity.
- TF-IDF is a commonly used representation in application scenarios such as information retrieval and text classification.

Pros and Cons of TFIDF

- We can use the TF-IDF vectors to calculate similarity between two texts using a similarity measure like Euclidean distance or cosine similarity.
- TF-IDF is a commonly used representation in application scenarios such as information retrieval and text classification.
- However, despite the fact that TF-IDF is better than the vectorization methods we saw earlier in terms of capturing similarities between words, it still suffers from the curse of high dimensionality.

Embeddings

Distributionally simliar words are words that are likely to occur in similar contexts.

- If we're given the word "USA," distributionally similar words could be other countries (e.g., Canada, Germany, India, etc.) or cities in the USA.
- If we're given the word "beautiful," words that share some relationship with this word (e.g., synonyms, antonyms) could be considered distributionally similar words.

Embeddings

Distributionally simliar words are words that are likely to occur in similar contexts.

- If we're given the word "USA," distributionally similar words could be other countries (e.g., Canada, Germany, India, etc.) or cities in the USA.
- If we're given the word "beautiful," words that share some relationship with this word (e.g., synonyms, antonyms) could be considered distributionally similar words.

Modern day NLP is based text representations which **learn** such semantic relationships in a dense, low dimensional space (compared to sparse, high dimensional space we saw earlier)

Word Embeddings

- pre-trained: such embedding representations are already learnt by training on a large database such as wikipedia. We can download the learnt models and directly use them.
- e.g., word2vec, glove, fastText etc.

```
from gensim.models import Word2Vec, KeyedVectors
pretrainedpath = "NLPBookTut/GoogleNews-vectors-negative300.bin"
w2v_model = KeyedVectors.load_word2vec_format(pretrainedpath
              , binary=True)
print('done loading Word2Vec')
print(len(w2v_model.vocab))
#Number of words in the vocabulary.
print(w2v_model.most_similar['beautiful'])
W2v_model['beautiful']
```

`most_similar('beautiful')` returns the most similar words to the word "beautiful." The output is shown below. Each word is accompanied by a similarity score. The higher the score, the more similar the word is to the query word:

```
[('gorgeous', 0.8353004455566406),
 ('lovely', 0.810693621635437),
 ('stunningly_beautiful', 0.7329413890838623),
 ('breathtakingly_beautiful', 0.7231341004371643),
 ('wonderful', 0.6854087114334106),
 ('fabulous', 0.6700063943862915),
 ('loveliest', 0.6612576246261597),
 ('prettiest', 0.6595001816749573),
 ('beatiful', 0.6593326330184937),
 ('magnificent', 0.6591402292251587)]
```

`w2v_model` returns the vector for the query word. For the word "beautiful," we get the vector as shown in Figure 3-6.

Document Embedding

- We can obtain the vector representation for an entire text by averaging individual word vectors.

```
import spacy
import en_core_web_sm

# Load the spacy model. This takes a few seconds.
nlp = en_core_web_sm.load()

# Process a sentence using the model
doc = nlp("Canada is a large country")

#Get a vector for individual words
#print(doc[0].vector) #vector for 'Canada', the first word
print(doc.vector) #Averaged vector for the entire sentence
```

The OOV problem

- We can also train our own word embeddings, but both pre-trained and self-trained word embeddings depend on the vocabulary they see in the training data.
- What should we do when we encounter a new word in a document?

The OOV problem

- We can also train our own word embeddings, but both pre-trained and self-trained word embeddings depend on the vocabulary they see in the training data.
- What should we do when we encounter a new word in a document?
- A simple approach that often works is to exclude those words from the feature extraction process so we don't have to worry about how to get their representations.
- Another way to deal with the OOV problem for word embeddings is to create vectors that are initialized randomly, where each component is between –0.25 to +0.25, and continue to use these vectors.
- There are also other approaches that handle the OOV problem by modifying the training process by bringing in characters and other subword-level linguistic components.

Beyond Words: Contextual representations

- In all the representations we've seen so far, we notice that one word gets one fixed representation. Can this be a problem?
- Well, to some extent, yes. Words can mean different things in different contexts.
- For example, the sentences "I went to a bank to withdraw money" and "I sat by the river bank and pondered about text representations" both use the word "bank." However, they mean different things in each sentence.
- Well, to some extent, yes. Words can mean different things in different contexts. For example, the sentences "I went to a bank to withdraw money" and "I sat by the river bank and pondered about text representations" both use the word "bank." However, they mean different things in each sentence.

SOTA: Universal Text Representations

- Neural architectures such as recurrent neural networks (RNNs) and transformers were used to develop large-scale models of language (BERT, and beyond), which can be used as pre-trained models to get text representations.
- Process: Take a large pre-trained model, and "fine-tune" it to a given task/dataset.
- These models have shown significant improvements on some fundamental NLP tasks, such as question answering, semantic role labeling, named entity recognition, and coreference resolution, to name a few.
- There are a lot of custom models, especially for BERT, such as: SciBERT, LAWBert, FinBERT, PatentBERT, BioBERT etc.

Conclusion

- we saw different techniques for representing text, starting from the basic approaches to state-of-the-art DL methods.
- When should we use what?

Conclusion

- we saw different techniques for representing text, starting from the basic approaches to state-of-the-art DL methods.
- When should we use what?
- For some applications, such as text classification, it's more common to see vectorization approaches and embeddings as the go-to feature representations for text.
- For some other applications, such as information extraction, it's more common to look for handcrafted, domain-specific features.
- Quite often, a hybrid approach that combines both kinds of features are used in practice.
- Having said that, BoW, BoN, TFIDF approaches are a great starting point!

Resources

Jupyter notebooks with code examples showing different ways of representing text: `https://github.com/practical-nlp/practical-nlp/tree/master/Ch3`