Feature Extraction and Word Embeddings

by Nitin Shelke

Bag-of-words vectorization

Whenever we apply any algorithm to textual data, we need to convert the text to a numeric form. Hence, there arises a need for some pre-processing techniques that can convert our text to numbers. Both bag-of-words (BOW) and TFIDF are pre-processing techniques that can generate a numeric form from an input text.

Bag-of-Words:

The bag-of-words model converts text into fixed-length vectors by counting how many times each word appears.

Whenever we apply any algorithm to textual data, we need to convert the text to a numeric form. Hence, there arises a need for some pre-processing techniques that can convert our text to numbers. Both bag-of-words (BOW) and TFIDF are pre-processing techniques that can generate a numeric form from an input text.

Let us illustrate this with an example. Consider that we have the following sentences:

DOC1: Text processing is necessary.

DOC2: Text processing is necessary and important.

DOC3: Text processing is easy.

If we take out the unique words in all these sentences, the vocabulary will consist of these 7 words: {'Text', 'processing', 'is', 'necessary', 'and', 'important, 'easy'}.

To carry out bag-of-words, we will simply have to count the number of times each word appears in each of the documents.

Document	Text	preprocessing	is	necessary	and	important	easy
1	1	1	1	1	0	0	0
2	1	1	1	1	1	1	0
3	1	1	1	0	0	0	1

Hence, we have the following vectors for each of the documents of fixed length -7:

Document 1: [1,1,1,1,0,0,0]

Document 2: [1,1,1,1,1,1,0]

Document 3: [1,1,1,0,0,0,1]

Limitations of Bag-of-Words:

If we deploy bag-of-words to generate vectors for large documents, the vectors would be of large sizes and would also have too many null values leading to the creation of sparse vectors.

Bag-of-words does not bring in any information on the meaning of the text. For example, if we consider these two sentences – “Text processing is easy but tedious.” and “Text processing is tedious but easy.” – a bag-of-words model would create the same vectors for both of them, even though they have different meanings.

In [ ]:
```
# Example 1
from sklearn.feature_extraction.text import CountVectorizer
corpus = ['Text processing is necessary.', 'Text processing is necessary and important.
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names())
```

In [ ]:
```
print(X.toarray())
```

In [ ]:
```
# Example 2
text = ["i love NLP",
        "NLP is future",
        "i will learn in 2 months"]
vectorizer = CountVectorizer()
count_matrix = vectorizer.fit_transform(text)
count_array = count_matrix.toarray()
df = pd.DataFrame(data=count_array,columns = vectorizer.get_feature_names())
print(df)
```

TFIDF Vectorization

TFIDF works by proportionally increasing the number of times a word appears in the document but is counterbalanced by the number of documents in which it is present. Hence, words like ‘this’, ’are’ etc., that are commonly present in all the documents are not given a very high rank. However, a word that is present too many times in a few of the documents will be given a higher rank as it might be indicative of the context of the document.

Term Frequency:

Term frequency is defined as the number of times a word (i) appears in a document (j) divided by the total number of words in the document.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

Inverse Document Frequency:

Inverse document frequency refers to the log of the total number of documents divided by the number of documents that contain the word. The logarithm is added to dampen the importance of a very high value of IDF.

$$idf(w) = log(\frac{N}{df_t})$$

TFIDF

It is computed by multiplying the term frequency with the inverse document frequency.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

Let us now see an illustration of TFIDF in the following sentences, that we refer to as documents.

Document 1: Text processing is necessary.

Document 2: Text processing is necessary and important.

Word	TF		IDF	TFIDF	
	Doc 1	Doc 2		Doc 1	Doc 2
Text	1/4	1/6	log (2/2) = 0	0	0
Processing	1/4	1/6	log (2/2) =0	0	0
Is	1/4	1/6	log (2/2) =0	0	0
Necessary	1/4	1/6	log (2/2) =0	0	0
And	0/4	1/6	log (2/1) =0.3	0	0.05
Important	0/4	1/6	log (2/1) =0.3	0	0.05

The above table shows how the TFIDF of some words are zero and some words are non-zero depending on their frequency in the document and across all documents.

The limitation of TFIDF is again that this vectorization doesn't help in bringing in the contextual meaning of the words as it is just based on the frequency.

In [ ]:
```
# TFIDF Vectorization code

#Example 1
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
print(X.toarray())

# The above array represents the vectors created for our 3 documents using the TFIDF vec
```

In [ ]:
```
# Example 2
import pandas as pd
text = ["i love the NLP",
        "NLP is the future",
        "i will learn the NLP"]
vectorizer = TfidfVectorizer()
matrix = vectorizer.fit_transform(text)
count_array = matrix.toarray()
df = pd.DataFrame(data=count_array,columns = vectorizer.get_feature_names())
print(df)
```

"NLP", "the" came in all the three documents hence it has a smaller vector value. "love" has a higher vector value since it appeared only once in a document.

Word Embedding

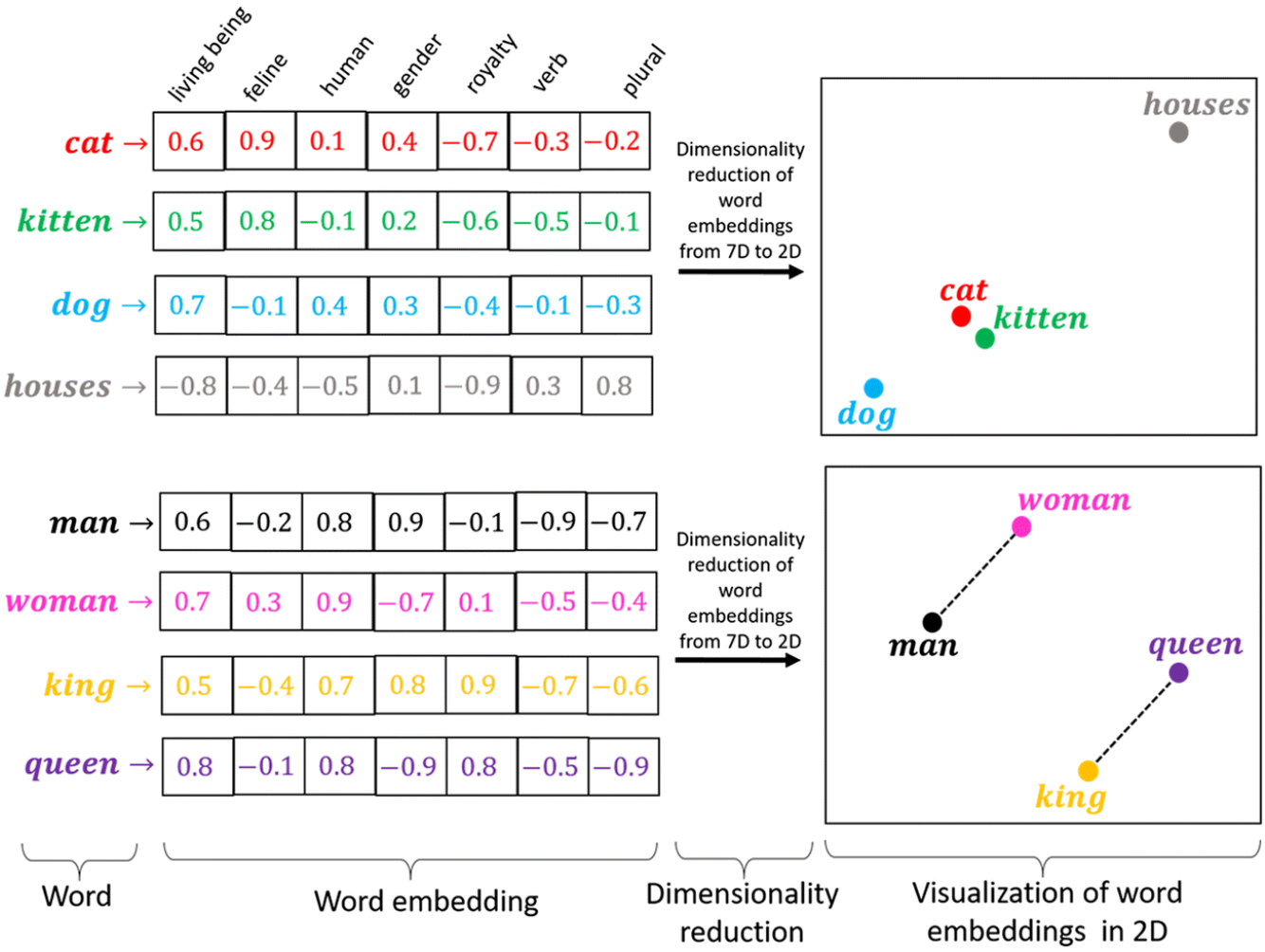
Imagine I have 2 words “love” and “like”, these two words have almost similar meanings but according to TF-IDF and BOW model these two will have separate feature values and these 2 words will be treated completely different.

TF-IDF, BOW model completely depends on the frequency of occurrence, it doesn’t take the meaning of words into consideration, hence above-discussed techniques are failed to capture the context and meaning of sentences.

“I like you” and “I love you” will have completely different feature vectors according to TF-IDF and BOW model, but that’s not correct.

note: while working with some classification task we would have big raw data and if we keep considering every synonym as a different word it will generate humongous numbers of tokens and this will cause numbers of features to get out of control.

Word embedding is a feature learning technique where words are mapped to vectors using their contextual hierarchy. similar words will have identical feature vectors.



As you notice, cats and kitten are placed very closely since they are related.

word embedding is trained on more than 6 billion words using shallow neural networks.

word2vec has 2 algorithms

1/CBOW

2/Skip-Gram

Implementing Glove word embedding using python:

don’t worry we don’t need to train word2vec, we will use pre-trained word vectors.

A pre-trained word vector is a text file containing billions of words with their vectors. we only need to map words from our data with the words in the word vector in order to get the vectors.

Pre-trained word vector file come in (50,100,200,300) dimension. here dimension is the length of the vector of each word in vector space. more dimension means more information about that word but bigger dimension takes longer time for model training.

In [ ]:
```
# Loading glove word embedding of 100 dimensions into a dictionary:

import numpy as np
glove_vectors = dict()
file = open('glove.6B.100d.txt', encoding = 'utf-8')
for line in file:
    values = line.split()
    word = values[0]
    vectors = np.asarray(values[1:])
    glove_vectors[word] = vectors
file.close()
```

In [ ]:
```
#we can use the get() method to glove_vectors to get the vectors
glove_vectors.get('house')
```

In [ ]:
```
# Creating a function that takes every sentence and returns word vectors:
vec_dimension = 100
def get_vec(x):
    arr  = np.zeros(vec_dimension)
    text = str(x).split()
    for t in text:
        try:
            vec = glove_vectors.get(t).astype(float)
            arr = arr + vec
        except:
            pass
    arr = arr.reshape(1,-1)[0]
    return(arr/len(text))
```

In [ ]:
```
x = ['I love you',
     'I love NLP and i will try to learn',
    'this is word embedding']
features = get_vec(x)
features
```

In [ ]: