

YouTube tutorial: <https://www.youtube.com/watch?v=RpWeNzfSUHw&t=323s>  
Theory: <https://machinelearningmastery.com/gentle-introduction-bag-words-model/>

**Bag-of-Words**

The bag-of-words model is a way of representing text data when modelling text with machine learning algorithms. It is used for feature extraction in [natural language processing](https://machinelearningmastery.com/natural-language-processing/).

A problem with modelling text is that it is messy, and techniques like machine learning algorithms prefer well defined fixed-length inputs and outputs.

Machine learning algorithms cannot work with raw text directly; the text must be converted into numbers. Specifically, vectors of numbers.

A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

1. A vocabulary of known words.
2. A measure of the presence of known words.

*A very common feature extraction procedures for sentences and documents is the bag-of-words approach (BOW). In this approach, we look at the histogram of the words within the text, i.e. considering each word count as a feature.*

**Bag-of-Words procedures**

Step 1. Collect Data

In this example, we shall use the following excerpt

*“It was the best of times,  
it was the worst of times,  
it was the age of wisdom,  
it was the age of foolishness,”*

Step 2. Design the vocabulary

We make a list of all the words in our model vocabulary. Ignoring case and punctuation, the unique words are:

* “it”
* “was”
* “the”
* “best”
* “of”
* “times”
* “worst”
* “age”
* “wisdom”
* “foolishness”

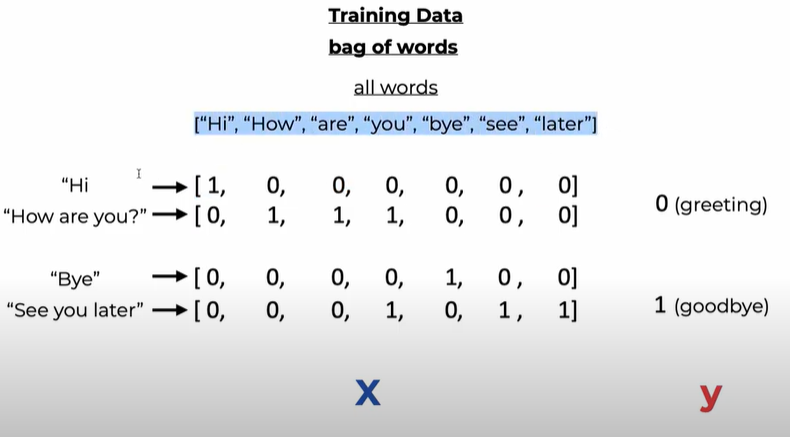
That is a vocabulary of 10 words from a corpus containing 24 words.

Step 3. Create Document Vectors

The next step is to score the words in each document.   
The objective is to turn each document of free text into a vector that we can use as input or output for a machine learning model.

Because we know that the vocabulary has 10 words, we can use a fixed-length document representation of 10, with one position in the vector to score each word.

The simplest scoring method is to mark the presence of words as a Boolean value – 0 for absent, 1 for present.



The scoring of the documents would look like this:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | “it was the best of times” = [1 ,1 ,1 ,1 ,1 ,1 ,0 ,0 ,0 ,0]  "it was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]  "it was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]  "it was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1] | |

All ordering of the words is nominally discarded and we have a consistent way of extracting features from any document in our corpus, ready for use in modelling.

New documents that overlap with the vocabulary of known words, but may contain words outside of the vocabulary, can still be encoded, where only the occurrence of known words are scored and unknown words are ignored. You can see how this might naturally scale to large vocabularies and larger documents.

**Managing Vocabulary**

As vocabulary size increases, so does the vector representation of documents. For a very large corpus, the length of the vector might be thousands or millions of positions. Further, each document may contain very few of the known words in the vocabulary.

This results in a vector with lots of zero cases, called a sparse vector or sparse representation.

Sparse vectors require more memory and computational resources when modelling and the vast number of positions or dimensions can make the modelling process very challenging for traditional algorithms.

As such, these is a need to decrease the size of the vocabulary when using a bag-of-words model. Simple text cleaning techniques include:

* Ignoring case
* Ignoring punctuation
* Ignoring frequent words that don’t contain much information, called stop words, like “a,” “of,” etc.
* Fixing misspelled words.
* Reducing words to their stem (e.g. “play” from “playing”) using stemming algorithms.

N-Grams (<https://blog.xrds.acm.org/2017/10/introduction-n-grams-need/>)

An N-Gram is simply a sequence of N-words, for example:

1. San Francisco (is a 2-gram)
2. The Three Musketeers (is a 3-gram)
3. She stood up slowly (is a 4-gram)

If we assign a probability to the occurrence of an N-gram of the probability of a word occurring next in a sequence of words, it can be very useful. Why?

First of all, it can help in deciding which N-grams can be chunked together to form single entities -such as “San Francisco” chunked together as one word, “high school” being chunked as one word.   
It can also help make next word predictions – say you have the partial sentence “Please hand over your…”, then it is more likely that the next word is going to be “test” or “assignment” or “paper” than it to be “school”.  
It can also help to make spelling error corrections. For instance, the sentence “drink cofee” could be corrected to “drink coffee” if you knew that the word “coffee” has a high probability of occurrence after the word “drink” and also the overlap of letters between “cofee” and “coffee” is high.

**Basically, an N-gram model predicts the occurrence of a word based on the occurrence of its N – 1 previous words. So here we are answering the question – how far back in the history of a sequence of words should we go to predict the next word? For instance, a bigram model (N = 2) predicts the occurrence of a word given only its previous word (as N – 1 = 1 in this case). Similarly, a trigram model (N = 3) predicts the occurrence of a word based on its previous two words (as N – 1 = 2 in this case).**

EXAMPLE

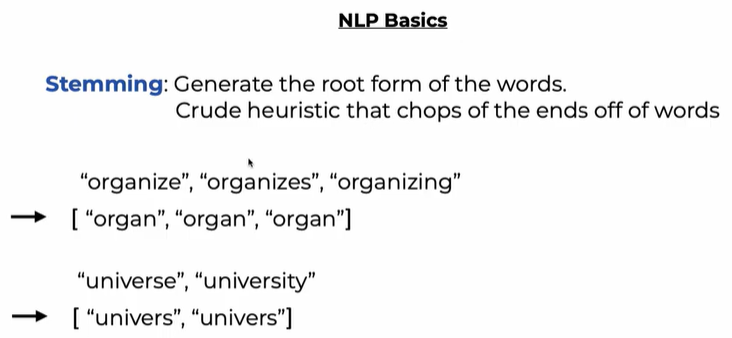
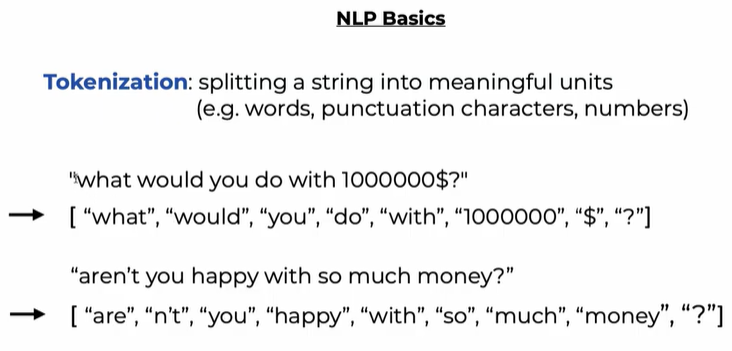
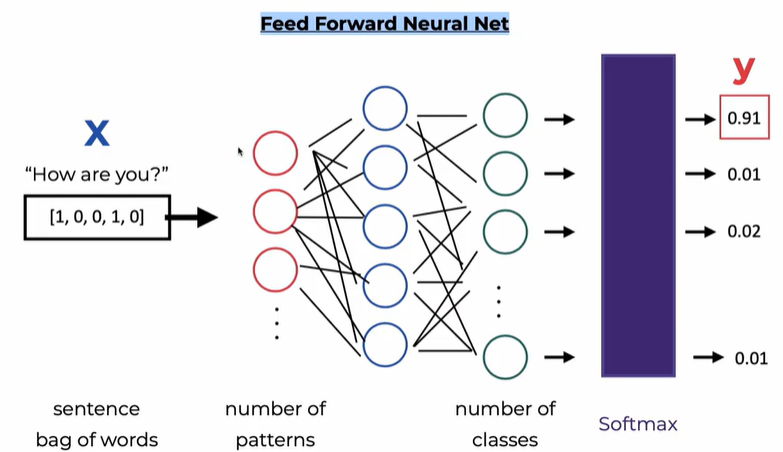
Consider the following sentences in our corpus:

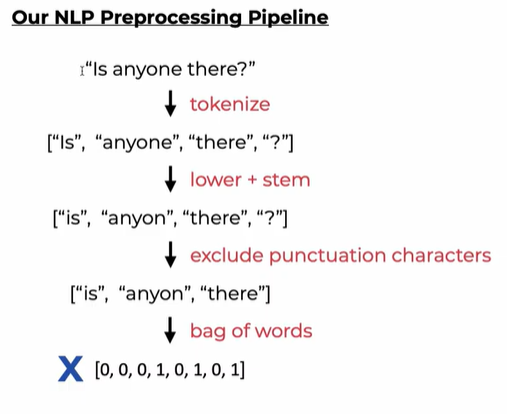
1. He said thank you.
2. He said bye as he walked through the door.
3. He went to San Diego.
4. San Diego has nice weather.
5. It is raining in San Francisco.

Assuming a bigram model, we are going to find the probability of a word based only on its previous word. In general, we can say that this probability is :

|  |
| --- |
| (the number of times the previous word ‘wp’ occurs before the word ‘wn’) / (the total number of times the previous word ‘wp’ occurs in the corpus) =  (Count (wp wn))/(Count (wp))  =(No. of times “Thank You” occurs) / (No. of times “Thank” occurs)  = 1/1  = 1  =(No of times “San Diego” occurs) / (No. of times “San” occurs)  = 2/3  = 0.67 |

Generally, the bigram model works well and it may not be necessary to use trigram models or higher N-gram models.





HOW TO USE THIS:

1. Ensure you have chatbot\_main, model\_chatbot, nltk\_utils and train\_chatbot in the same folder.  
2. nltk\_utils contains utility functions and contains the baseline Bag-of-Words model

BoW: <https://machinelearningmastery.com/gentle-introduction-bag-words-model/>

3. model\_chatbot contains the neural network required to train the chatbot

3a. Load the ‘intents.json’ file

3b. Get the patterns in each intends and tokenize it

3c. X\_train contains the bag of words, Y\_train contains the labels

3d. The neural network will train the bag of words to find the correct labels and produce the loss

4. train\_chatbot trains the chatbot and generates a data.pth file, which will be used by chatbot\_main as the model

5. Run the programme using chatbot\_main

6. New responses can be added in by adding it into the intents.json file – remember to run train\_chatbot again to update the data.pth file!