Introduction to Natural Language Processing

Module 2 | Chapter 4 | Notebook 4

The SVM you got to know as a classifier in the last chapter, is often used in spam filters due to its fast predictions, as quick decisions are a deciding factor in this situation. Before we build a spam filter in the context of a given scenario, we should first understand how computers can use human language. Therefore, in this lesson we'll learn some basic word processing techniques used in Natural Language Processing (NLP). You will learn how to:

- Prepare text data for text classification tasks.
- Use different Python modules for preparing text data.

Natural Language Processing

Natural Language Processing, NLP for short, is a branch of computer science and artificial intelligence, which deals with the interaction between computers and human (natural) languages.

Many of the technologies we know are based on NLP applications. For example, a lot of smartphones and online search engines have text prediction features that can complete what you are typing. They do this based on the letters or words you have already typed. Online translation tools are based on NLP techniques to translate words from one language to another. Automatic spellchecking is also the result of NLP.

In this chapter you will learn about a popular NLP business application: text tagging (also known as text categorization). The goal of text classification models is to automatically assign texts to one or more categories.

Scenario: A company from Singapore gets in touch with you. They want to improve their customer service by giving their customers the opportunity to contact them by text message. However, they're afraid that they'll receive a lot of spam messages, so they want to detect and filter them automatically. Your task is to build a first prototype and make a *proof of concept*. Since the company is at the very beginning of their project, they don't have their own data set, and are working with data they have purchased.

Let's start by taking a look at the data. Open the file *text_messages.csv* as a `DataFrame` named `df`, use the `'Unnamed: 0'` column for `index_col` and print the first 5 lines.

In [1]:
```
import pandas as pd
df = pd.read_csv('text_messages.csv', index_col=0)
df.head()
```

Out[1]:

	status	msg
0	0	Go until jurong point, crazy.. Available only ...
1	0	Ok lar... Joking wif u oni...
2	1	Free entry in 2 a wkly comp to win FA Cup fina...
3	0	U dun say so early hor... U c already then say...
4	0	Nah I don't think he goes to usf, he lives aro...

`df` has two columns: `'msg'`, which contains the text messages, and `'status'`, which indicates whether it is a normal text message (`'status'` = `0`) or whether it is spam (`'status'` = `1`). In NLP this kind of collection of text data is called a **corpus**.

Let's take a closer look at the data. Run the cell below to see two text messages from the data set. Which is a normal text message and which is spam?

In [2]:
```
print(df.loc[122, 'msg'])
print(df.loc[105, 'msg'])
```

```
Todays Voda numbers ending 7548 are selected to receive a $350 award. If you have a m
atch please call 08712300220 quoting claim code 4041 standard rates app
Thanks a lot for your wishes on my birthday. Thanks you for making my birthday truly
memorable.
```

You probably already noticed that the first message is probably spam and the second message is a normal text message. Real messages are usually sent consciously from one person to another and have a certain similarity with verbal conversations. Spam messages are often unwanted and often contain advertising for a product.

How can a computer tell the difference? In supervised learning, you use a combination of features to create models for classifying data into categories. But `df` itself has no other columns except the text messages and the target variable to be predicted (`status`).

This is where NLP comes into play. First of all, the text data is cleaned up to correct errors and to give it more structure for analysis (*data cleaning and preparation*). Then NLP methods are used to obtain certain parts of the text which reflect the categories as well as possible. These characteristics can then be used with classification models.

To clean and prepare the data, we'll use a combination of functions and methods from modules in standard Python libraries as well as modules specially developed for NLP.

In the following cell, import the modules `string` and `re` (short for *regular expressions*). These modules offer a lot of useful functions and methods for working with strings

In [3]:
```
import string
import re
```

There is a range of NLP modules in Python. In this chapter we'll use `spacy` (see official documentation) and `nltk` (see official documentation). Import both these modules in the cell below.

If you see an error message saying `Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or directory`, you can ignore it. It has to do with the fact that no graphics card (GPU) is installed on the server (`Ignore above cudart dlerror if you do not have a GPU set up on your machine.`).

In [4]:
```python
import spacy
import nltk
```

```
2024-04-24 14:38:56.308199: W tensorflow/stream_executor/platform/default/dso_loader.
cc:60] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.
0: cannot open shared object file: No such file or directory
2024-04-24 14:38:56.308234: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Igno
re above cudart dlerror if you do not have a GPU set up on your machine.
```

`spacy` and `nltk` offer similar tools for NLP, but have they different approaches.

Functions from `nltk` accept `str` values and output `str` values. In contrast, `spacy` uses an object-oriented approach and usually returns **document objects** with their own attributes and methods. Many users find `spacy` to be more time and memory efficient than `nltk` and therefore more suitable for production.

For this reason, we will use `spacy` to perform most NLP tasks for cleaning text data, with the help of certain functions and methods from `nltk`, `re` and `string`.

To do this, we need a statistical model from `spacy`. Your choice of model should depend on the language of the corpus you want to analyze. You can find the complete list here.

Import `sentences` from `spacy.lang.en.examples` and load the standard model for English `en_core_web_sm` as `str` with the function `spacy.load()`. Store the result in the variable `nlp` and print the data type of this variable.

In [8]:
```python
from spacy.lang.en.examples import sentences
nlp = spacy.load('en_core_web_sm')
print(type(nlp))
```

```
<class 'spacy.lang.en.English'>
```

You can see that `nlp` has the data type `Language`. This means that it contains all components necessary for processing English text.

Congratulations: You've just learned what natural language processing is, what tasks are involved and which Python modules can help you. You also loaded the English statistical model of `spacy` to enable NLP tasks for an English corpus. Next, you'll learn how text data is cleaned and prepared (*preprocessing*) for further use in machine learning algorithms.

Tokenization

In order to analyze text data correctly, machine learning models should be able to recognize structures in the text, such as individual words and their language components. We achieve this by what's called **tokenization**. Here, the corpus is broken down into **significant linguistic units**, such as words or sentences, and saved as a list. The elements of this list are called *tokens*.

In our example, it makes sense to separate the data into individual words, since we already have very short text messages. For other tasks, however, it may well be better to break the text down into letters or word fragments.

To do this, we must first create a `spacy` specific `.Doc` object containing the text in the corpus. To do this, we'll use our `Language` object `nlp` and pass it the text that we want to analyze. Run the following cell to split one of the messages in the data set into *tokens*:

In [9]:
```
# print original message
print(df.loc[105, 'msg'])
print(type(df.loc[105, 'msg']))

# create a doc variable
doc = nlp(df.loc[105, "msg"])
print(doc)

# check data type of doc
type(doc)
```

```
Thanks a lot for your wishes on my birthday. Thanks you for making my birthday truly
memorable.
<class 'str'>
Thanks a lot for your wishes on my birthday. Thanks you for making my birthday truly
memorable.
```

Out[9]:
```
spacy.tokens.doc.Doc
```

You can see that the text has been preserved, but the `type` has been changed. A `.Doc` object is a sequence of `token` objects, which are the individual linguistic units we need for our analysis. We can iterate through them with a `for` loop. The text of a `token` is in the attribute `my_token.text`.

In the following cell, we'll use a list comprehension to generate a list of the text contents of each `token`:

In [10]:
```
doc_tokens = [token.text for token in doc]
print(doc_tokens)
```

```
['Thanks', 'a', 'lot', 'for', 'your', 'wishes', 'on', 'my', 'birthday', '.', 'Thank
s', 'you', 'for', 'making', 'my', 'birthday', 'truly', 'memorable', '.']
```

`doc_tokes` is now divided into words and punctuation marks, while `spacy` recognizes individual `tokens` mainly by separating them with spaces. However, there are some exceptions: For example, contractions of two words in English like "*I'll*" would be tokenized as "I" and "*will*". You can learn more about how tokenization works with `spacy` here.

The code below stores the 123rd text message in `df` as a `.Doc` object named `other_doc`:

In [25]:
```
# print original message
print(df.loc[122, 'msg'])

# create doc variable from message
other_doc = nlp(df.loc[122, 'msg'])
```

```
Todays Voda numbers ending 7548 are selected to receive a $350 award. If you have a m
atch please call 08712300220 quoting claim code 4041 standard rates app
```

Divide the text in `other_doc` into individual tokens. Store it as a list named `other_doc_tokens`.

In [12]:
```
doc_tokens = [token.text for token in other_doc]
print(doc_tokens)
```

```
['Todays', 'Voda', 'numbers', 'ending', '7548', 'are', 'selected', 'to', 'receive',
'a', '$', '350', 'award', '.', 'If', 'you', 'have', 'a', 'match', 'please', 'call',
'08712300220', 'quoting', 'claim', 'code', '4041', 'standard', 'rates', 'app']
```

`spacy` not only divides the message up, but also performs *part of speech tagging* (POS tagging, see here) This means that for each `token`, the type of word is indicated, such as nouns (`'NOUN'`) or punctuation (`'PUNCT'`).

We can access this with the `my_token.pos_` attribute.

In [13]:
```
token_pos = [[token.text, token.pos_] for token in doc]
print(token_pos)
```

```
[['Thanks', 'NOUN'], ['a', 'DET'], ['lot', 'NOUN'], ['for', 'ADP'], ['your', 'PRON'],
['wishes', 'NOUN'], ['on', 'ADP'], ['my', 'PRON'], ['birthday', 'NOUN'], ['.', 'PUNC
T'], ['Thanks', 'NOUN'], ['you', 'PRON'], ['for', 'ADP'], ['making', 'VERB'], ['my',
'PRON'], ['birthday', 'NOUN'], ['truly', 'ADV'], ['memorable', 'ADJ'], ['.', 'PUNC
T']]
```

You can find the complete list of tags and their meaning here. These tags are based on a standard for POS tags that are defined in the Universal Dependencies project.

Create a list containing the text and POS tag of each `token` of `other_doc`. Call it `other_token_pos` and print it.

In [20]:
```
other_token_pos = [ [token.text, token.pos_] for token in other_doc]
other_token_pos
```

Out[20]:
```
[['Todays', 'NOUN'],
 ['Voda', 'PROPN'],
 ['numbers', 'NOUN'],
 ['ending', 'VERB'],
 ['7548', 'NUM'],
 ['are', 'AUX'],
 ['selected', 'VERB'],
 ['to', 'PART'],
 ['receive', 'VERB'],
 ['a', 'DET'],
 ['$', 'SYM'],
 ['350', 'NUM'],
 ['award', 'NOUN'],
 ['.', 'PUNCT'],
 ['If', 'SCONJ'],
 ['you', 'PRON'],
 ['have', 'VERB'],
 ['a', 'DET'],
 ['match', 'NOUN'],
 ['please', 'INTJ'],
 ['call', 'VERB'],
 ['08712300220', 'NUM'],
 ['quoting', 'VERB'],
 ['claim', 'NOUN'],
 ['code', 'NOUN'],
 ['4041', 'NUM'],
 ['standard', 'ADJ'],
 ['rates', 'NOUN'],
 ['app', 'PROPN']]
```

Congratulations: You have just learned how to create a `Doc` object with `spacy` to split text data into `token` objects and perform POS tagging. In the next section, you'll use the `.Doc` object to clean up and format the text data.

Cleaning and preparing text data

The tokenization process is then followed by text cleaning. Text data can take various forms: in books, magazines, websites, online news, etc. The length of the text also varies: Text messages are usually much shorter than novels. In order for our models to handle them, all these text forms first have to be prepared and converted into a format consisting of numbers. But before we do this, we want to make an adjustment. This is necessary because a computer cannot really speak and therefore a lot of aspects of human language, which are very important for us humans, are irrelevant for an algorithm.

Some cleaning and preparation techniques are:

1. **Lemmatization**
2. **Removing stop words**
3. **Removing punctuation marks**

Lemmatization means that words are reduced to their **root form**, also known as a *lemma*. For grammatical reasons, different forms of the same word can appear in a text, for example *make*, *makes*, *making* or *maker*. Python would consider these variations of the word *make* as separate words, even though their meaning is the same. Lemmatization transforms all variations of *make* to this root form.

With the `my_token.lemma_` attribute, you can obtain the root form of the corresponding `token`. POS tagging is used to determine the appropriate lemma. Execute the following cell to get the basic forms of the words in `doc` as a list (make sure that `doc` is a `.Doc` object).

In [38]:
```
# print original message
print("ORIGINAL TEXT:\n", doc)

# get the lemmas of doc
lemma_token = [token.lemma_ for token in doc]
print("AFTER LEMMATIZATION:\n", lemma_token)
```

```
ORIGINAL TEXT:
 Thanks a lot for your wishes on my birthday. Thanks you for making my birthday truly
memorable.
AFTER LEMMATIZATION:
 ['thank', 'a', 'lot', 'for', 'your', 'wish', 'on', 'my', 'birthday', '.', 'thank',
'you', 'for', 'make', 'my', 'birthday', 'truly', 'memorable', '.']
```

Have you noticed how, for example, "Thanks", "wishes" and "making" were lemmatized and that all the words are now lower case?

Pronouns are tagged in `spacy` as `'PRON'` via the attribute `my_token.pos_`. If you don't want them in the list, you can add an `if` statement to the list comprehension. Let's modify the `lemma_token` list, which contains the token lemmas as strings, and overwrite it.

In [39]:
```
lemma_token = [token.lemma_ for token in doc if token.pos_ != "PRON"]
print('\n')
print(lemma_token)
```

```

['thank', 'a', 'lot', 'for', 'wish', 'on', 'birthday', '.', 'thank', 'for', 'make',
'birthday', 'truly', 'memorable', '.']
```

Use a list comprehension to get the basic forms of the `token` of the text message in `other_doc` as a list. Make sure you remove the pronouns using the `my_token.pos_` attribute. You can assign the result to the variable `other_lemma_token`.

In [27]:
```
other_lemma_token = [token.lemma_ for token in other_doc if token.pos_ != "PRON"]
print('\n')
print(other_lemma_token)
```

```

['today', 'Voda', 'number', 'end', '7548', 'be', 'select', 'to', 'receive', 'a', '$',
'350', 'award', '.', 'if', 'have', 'a', 'match', 'please', 'call', '08712300220', 'qu
ote', 'claim', 'code', '4041', 'standard', 'rate', 'app']
```

Removing stop words is a way to remove common words from a text. Stop words are generally articles ("the" and "a"), pronouns such as "I" and "you" (which we already removed in the previous step) or common verbs ("be", "can"). These words appear frequently in most English language texts. Removing these words reduces the amount of data that needs to be analyzed, while allowing machine learning algorithms to put more emphasis on tokens that give a text its true meaning.

For this exercise we will use the stop words provided in the `nltk` module. Run the following cell to import, store and print them. We receive the stop words as a `list`, which we can look at more closely. Import `stopwords` from the `nltk.corpus` module and store the English stop words, specified with `stopwords.words('english')`, in the variable `stopWords`. Then print `stopWords`.

In [28]:
```
# import stopwords from nltk
from nltk.corpus import stopwords

# convert stopwords to set
stopWords = stopwords.words('english')

# print stopwords
print(stopWords)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you'v
e", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',
'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'thi
s', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'bee
n', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an',
'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by',
'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before',
'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'ove
r', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'w
hy', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'su
ch', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's',
't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll',
'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "di
dn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'i
sn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'sh
an', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won',
"won't", 'wouldn', "wouldn't"]
```

We now want to remove all the stop words from `lemma_token`. There are several ways to do this. We'll be pragmatic and simply use a list comprehension again with an `if` statement. We will also replace all tokens by their lowercase version using `my_token.lower()`. Run the following cell:

In [29]:
```
# print message
print("LEMMATIZED TEXT:\n", lemma_token)

# remove stopwords and print again
no_stopWords_lemma_token = [token.lower() for token in lemma_token if token not in sto
print("NO STOP WORDS:\n", no_stopWords_lemma_token)
```

```
LEMMATIZED TEXT:
 ['thank', 'a', 'lot', 'for', 'wish', 'on', 'birthday', '.', 'thank', 'for', 'make',
'birthday', 'truly', 'memorable', '.']
NO STOP WORDS:
 ['thank', 'lot', 'wish', 'birthday', '.', 'thank', 'make', 'birthday', 'truly', 'mem
orable', '.']
```

The effect is immediately clear. Create a new list from `other_lemma_token` by ignoring the stop words using list comprehension. Store these as `other_no_stopWords_lemma_token`.

```
In [30]: # print message
         print("LEMMATIZED TEXT:\n", other_lemma_token)

         # remove stopwords and print again
         other_no_stopWords_lemma_token = [token.lower() for token in other_lemma_token if toke
         print("NO STOP WORDS:\n", other_no_stopWords_lemma_token)
```

```
LEMMATIZED TEXT:
 ['today', 'Voda', 'number', 'end', '7548', 'be', 'select', 'to', 'receive', 'a',
'$', '350', 'award', '.', 'if', 'have', 'a', 'match', 'please', 'call', '0871230022
0', 'quote', 'claim', 'code', '4041', 'standard', 'rate', 'app']
NO STOP WORDS:
 ['today', 'voda', 'number', 'end', '7548', 'select', 'receive', '$', '350', 'award',
'.', 'match', 'please', 'call', '08712300220', 'quote', 'claim', 'code', '4041', 'sta
ndard', 'rate', 'app']
```

Just like removing stop word, **removing punctuation** involves removing punctuation marks and symbols that do not contribute to the meaning of the text. We can use `punctuation` from the `string` module. This is a string made up of punctuation marks and symbols. We can remove these from our text just like the stop words.

```
In [31]: # save punctuations and print them
         punctuations = string.punctuation
         print(punctuations)
```

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

To remove the punctuation marks, we proceed in the same way as we did for stop words: We use a *list comprehension* with an `if` query. We can iterate through a string as if it were a list. Python then accesses the individual letters and only allows comparisons with strings. Since this is our last stage of text cleaning, we'll use a corresponding variable. Run the next cell.

```
In [32]: # print message
         print("TEXT WITH NO STOP WORDS:\n", no_stopWords_lemma_token)

         # remove punctuations
         clean_doc = [token for token in no_stopWords_lemma_token if token not in punctuations]
         print("NO PUNCTUATIONS:\n", clean_doc)
```

```
TEXT WITH NO STOP WORDS:
 ['thank', 'lot', 'wish', 'birthday', '.', 'thank', 'make', 'birthday', 'truly', 'mem
orable', '.']
NO PUNCTUATIONS:
 ['thank', 'lot', 'wish', 'birthday', 'thank', 'make', 'birthday', 'truly', 'memorabl
e']
```

Now the text message is cleaner, without punctuation marks and only contains the most important words of the text.

Now remove the punctuation marks from `other_no_stopWords_lemma_token`. Store the result as `clean_other_doc` and print it to check if the punctuation marks have been removed.

In [33]:
```
# print message
print("TEXT WITH NO STOP WORDS:\n", other_no_stopWords_lemma_token)

# remove punctuations
clean_other_doc = [token for token in other_no_stopWords_lemma_token if token not in p
print("NO PUNCTUATIONS:\n", clean_other_doc)
```

```
TEXT WITH NO STOP WORDS:
 ['today', 'voda', 'number', 'end', '7548', 'select', 'receive', '$', '350', 'award',
'.', 'match', 'please', 'call', '08712300220', 'quote', 'claim', 'code', '4041', 'sta
ndard', 'rate', 'app']
NO PUNCTUATIONS:
 ['today', 'voda', 'number', 'end', '7548', 'select', 'receive', '350', 'award', 'mat
ch', 'please', 'call', '08712300220', 'quote', 'claim', 'code', '4041', 'standard',
'rate', 'app']
```

Great! Now let's take a look at another message: The row with number `3202` in `df`. We'll practice cleaning text with this message. Run the following cell to get a first impression of the text.

In [34]:
```
print(df.loc[3202, 'msg'])
```

```
Haha... Yup hopefully  we will lose a few kg by mon. after hip hop can go orchard and
weigh again
```

Use the template below to guide you through the steps. Make sure you use the list comprehensions and `if` statements in the same order to clean up your `.Doc` variable. Use appropriate variables to create the result `clean_doc`.

In [40]:
```
# Creating a .Doc variable `doc` using `nlp()` for message with line number 3202
doc_new = nlp(df.loc[3202, "msg"])

# Lemmatization: creating a list of the attributes from my_token.lemma_
# Using an if-request in order to exclude 'PRON'
new_lemma_token = [token.lemma_ for token in doc_new if token.pos_ != "PRON"]

# Removing stopwords: creating a list without words that are contained in stopWords an
new_lemma_token = [token.lower() for token in new_lemma_token if token not in stopWord

# Removing punctuations: Creating a list without strings that are contained in punctua
new_lemma_token = [token for token in new_lemma_token if token not in punctuations]

# print clean_doc
print(new_lemma_token)
```

```
Haha... Yup hopefully  we will lose a few kg by mon. after hip hop can go orchard and
weigh again
['haha', '...', 'yup', 'hopefully', ' ', 'lose', 'kg', 'mon', 'hip', 'hop', 'go', 'or
chard', 'weigh']
```

After cleaning and preparing the text, you should have a `list` of strings that looks like this:

```
['haha', '...', 'yup', 'hopefully', ' ', 'lose', 'kg', 'mon', 'hip',
'hop', 'go', 'orchard', 'weigh']
```

Do you notice anything strange? There's an element that is just a space `' '` and a token with an ellipsis `'...'`. These tokens don't provide meaningful insights into the text. You'll learn how to deal with this in the next section.

Congratulations: You have just learned how to perform lemmatization, remove stop words and punctuation marks to clean up text data and prepare it for analysis. In the next section we'll combine all these techniques into a single function.

A user-defined function to clean and prepare text

In the previous section, you carried out at least 4 steps to clean and prepare the text. We want to apply this process to every message. So it makes sense to create a function specially for this task.

Follow the template for `text_cleaner()` below, which takes an `str` value - `sentence`. For the time being, the function should return a `list` of strings. Use the same steps as in the last code cell.

In [41]:
```
def text_cleaner(sentence):
    # Create the Doc object named `text` from `sentence` using `nlp()`
    doc = nlp(sentence)
    # Lemmatization
    lemma_token = [token.lemma_ for token in doc if token.pos_ != 'PRON']
    # Remove stop words and converting tokens to lowercase
    no_stopWords_lemma_token = [token.lower() for token in lemma_token if token not ir
    # Remove punctuations
    clean_doc = [token for token in no_stopWords_lemma_token if token not in punctuati
    # Output
    return clean_doc
```

Run the cell below to see whether your function works. The result should be the same as in the previous task:

```
['haha', '...', 'yup', 'hopefully', ' ', 'lose', 'kg', 'mon', 'hip',
'hop', 'go', 'orchard', 'weigh']
```

In [44]:
```
print(text_cleaner(df.loc[3202, "msg"]))
```

```
<class 'list'>
['haha', '...', 'yup', 'hopefully', ' ', 'lose', 'kg', 'mon', 'hip', 'hop', 'go', 'or
chard', 'weigh']
```

Now it's time to get rid of the space and the ellipsis. We'll also merge the individual strings in the list back into a single string, because we need the data in this format for the next processing steps.

Copy and paste your code from the last code cell and add two lines before the `return` statement.

1. Convert the return value (a `list` of strings) into a string using the `my_str.join` method (link for a short tutorial). It takes a list and links the elements with the string the function is called with. Use a single space to connect the elements.
2. Then use the `re.sub()` function from the `re` module. This allows you to replace multiple spaces or dots with a single space. According to the official documentation, this function takes 3 arguments: `re.sub(pattern, repl, string)`
   - `pattern` is the regular expression to be replaced. The regular expression we need is `'[\.\s]+'`. It means that at least one dot or space must be present.
   - `repl` is the value used to replace the matched expression. In our example we want to replace the expressions with a single space `' '`.
   - `string` is the text this substitution should be carried out on In our example this is the previous return value.

Tip: Neither `my_str.join()` nor `re.sub()` modify the object directly. So remember to reassign the result back to the variable name. Normally you should add a *docstring* to functions you write yourself. But you can leave that out this time.

In [51]:
```python
def text_cleaner(sentence):
    # Create the Doc object named `text` from `sentence` using `nlp()`
    doc = nlp(sentence)
    # Lemmatization
    lemma_token = [token.lemma_ for token in doc if token.pos_ != 'PRON']
    # Remove stop words and converting tokens to lowercase
    no_stopWords_lemma_token = [token.lower() for token in lemma_token if token not ir
    # Remove punctuations
    clean_doc = [token for token in no_stopWords_lemma_token if token not in punctuati

    # Use the `.join` method on `text` to convert string
    joined_clean_doc = " ".join(clean_doc)
    # Use `re.sub()` to substitute multiple spaces or dots`[\.\s]+` to single space `'
    final_doc = re.sub('[\.\s]+', ' ', joined_clean_doc)

    # Output
    return final_doc
```

You'll know that your function works if the result for the message at index `3202` for the following code looks like this:

```
'haha yup hopefully lose kg mon hip hop go orchard weigh'
```

In [52]:
```python
text_cleaner(df.loc[3202, "msg"])
```

Out[52]:
```
'haha yup hopefully lose kg mon hip hop go orchard weigh'
```

Congratulations: The company is pleased that you've made such rapid progress with data cleaning and have written a reusable function for it.

Remember:

- Text data first has to be cleaned and prepared before it can be analyzed.

- Some common text cleaning tasks for NLP are lemmatization, as well as removing stop words and punctuation.
- Some helpful modules for NLP are `spacy`, `nltk`, `string` and `re`.

The data in this chapter was used here first: Almeida, T.A., Gomez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.