Vectorizing Texts

Module 2 | Chapter 4 | Notebook 5

In this lesson we'll take the final step to turn texts into usable numerical objects for machine learning algorithms. This process is called **vectorization** (mathematically speaking, the final objects are vectors, the basic building blocks of linear algebra). By the end of this lesson you will be able to:

- What the *bag of words* and *term frequency - inverse document frequency* methods do with texts.
- How to use *bag of words* and *term frequency - inverse document frequency* with `sklearn`.
- What the difference is between these two methods.

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. You are supposed to present a first concept for it. 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.

Now that you have learned how to clean text data, we can start to display the text in numerical form. From there, we can generate functions to analyze a text with machine learning algorithms. The process of converting text to numbers for NLP is called **vectorization** and in this lesson you will learn two of the most popular vectorization techniques: *Bag of words* and *term frequency - inverse document frequency*.

Let's start by repeating the cleaning and preparation techniques you learned previously.

In the cell below, import the `pandas` module with its conventional name and store the data set *text_messages.csv* as a `DataFrame` named `df` and use the zeroth column for `index_col`. Then print the first five rows.

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...

Now import the modules that we want to use for executing NLP tasks, namely `spacy`, `nltk`, `string` and `re` (ignore the possible `dlerror` as in the previous exercise).

In [2]:
```python
import spacy
import nltk
import string
import re
```

```
2024-04-25 09:53:11.290167: 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
2024-04-25 09:53:11.290207: 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.
```

Next, we'll load the tools we need from these modules.

Now import `stopwords` from `nltk.corpus`. Then you get the english stop words by using `stopwords.words('english')` as `list`. Convert it to a `set` and save it as `stopWords`.

In [4]:
```python
from nltk.corpus import stopwords
stopWords = stopwords.words('english')
```

Store punctuation characters from `string.punctuation` as a variable named `punctuations`.

In [5]:
```python
punctuations =  string.punctuation
```

Run the following cell to initialize a `Language` class from `spacy`. We'll use the English language model `"en_core_web_sm"`.

In [6]:
```python
nlp = spacy.load('en_core_web_sm')
```

In the following code cell we'll the `text_cleaner` function from the last exercise again. Then we'll use it to clean up our messages.

In [7]:
```python
# `text_cleaner` function
def text_cleaner(sentence):
    """Clean the text using typical NLP-Steps.

    Steps include: Lemmatization, removing stop words, removing punctuations

    Args:
        sentence (str): The uncleaned text.

    Returns:
        str: The cleaned text.

    """

    # 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)
    return final_doc
```

Now it's time to apply our function to all the messages in `df`.

This is where `pandas` helps us with the `my_series.apply()` method - (link to the documentation) You pass a function to it, which is then applied to each element of the `Series` or the column of the `DataFrame`. It is important to note that this method **can only be applied to individual columns** of a `DataFrame`. The result is a `Series` of the corresponding return values of the function. You can find the documentation here.

Use `my_series.apply()` on the `'msg'` column of `df` and pass the `text_cleaner` function to it to apply it to all the text messages. Store the results in a new column in `df` with the name `'msg_clean'`. The execution of the code may take a little while. Then print the first 5 rows of `df`.

In [9]:
```
df.loc[:, 'msg_clean'] = df.loc[:, 'msg'].apply(text_cleaner)
df.head()
```

Out[9]:

	status	msg	msg_clean
0	0	Go until jurong point, crazy.. Available only ...	go jurong point crazy available bugis n great ...
1	0	Ok lar... Joking wif u oni...	ok lar joke wif u oni
2	1	Free entry in 2 a wkly comp to win FA Cup fina...	free entry 2 wkly comp win fa cup final tkts 2...
3	0	U dun say so early hor... U c already then say...	u dun say early hor u c already say
4	0	Nah I don't think he goes to usf, he lives aro...	nah think go usf live around though

Now you should see the following three columns in `df`: `'status',` 'msg' `and` 'msg_clean' `.` `In` 'msg_clean'` all the words should be written in lower case and only appear in their root form (*lemmatized*).

Now you've cleaned and prepared all the text messages. We can now start vectorizing the texts.

Since we only have one data set available in this exercise, we can train our model, but not test it on another data set. This is why in this kind of case you should split your data set into a training and test set as early as possible, especially as we are aiming to use the data to develop a spam filter. The `train_test_split()` function from `sklearn` provides us with the tool we need to do this - (link to documentation). `train_test_split()` splits the data set into a training data set and a test data set. By default, the data set is randomly spread between a training and test data set, but you should determine the size of the test data sets to be created from this:

```
train_test_split(X, #features (DataFrame or Series)
                 y, #target (DataFrame or Series)
                 test_size =float, #size of test set (between 0.1 and 1.0)
                 random_state=int) #random seed generator (enables
reproducibility)
```

It's important to note that the output of this function is assigned to the following variables again: `X_train`, `X_test`, `y_train`, `y_test`. It's important to note the order of variables here: First the features for the training set and then for the test set are created, then the target for the training set and finally the target for the test set. Always make sure that you stick to this order when assigning the output of `train_test_split()` to your features and your target vector, both for the training and the test dataset!

Import `train_test_split()` from `sklearn.model_selection` in the code cell below.

In [10]:
```
from sklearn.model_selection import train_test_split
```

Next, create the feature matrix with the name `features` from the `'msg_clean'` column with the cleaned text from `df` and a target vector with the name `target` from the `'status'` column. **Attention:** For once, the features must not be a DataFrame, but a Series. Otherwise there will be problems later on.

In [16]:
```
features = df.loc[:, 'msg_clean']
target  = df.loc[:, 'status']
```

Now create the training and test data sets and name them `features_train`, `features_test`, `target_train` and `target_test`. Use a value of `0.3` for `test_size` (remember that this assigns 30% of `features` to the test data set) and yse a value of `1` for `random_state` (to make our results reproducible).

In [19]:
```
features_train, features_test, target_train, target_test = train_test_split(features,
                                                                            target,
                                                                            test_size
                                                                            random_sta
```

Congratulations: You have just combined the text cleanup and editing techniques from the previous lesson again, applied your cleaning function to all the messages, and created a training set and a test set.

The *bag of words* method

The *bag of words* (BoW) method represents words in a **corpus** based on the frequency of the word in each text. A BoW algorithm takes all the unique words present in the corpus and displays each text based on how often a word appears in that text. The method gets its name from the fact that the order and structure of the words in the text are ignored. It only matters whether a word is contained in the text, and not where it is in the text.

To illustrate how the BoW method works, let's only concentrate on 2 messages in our corpus. First look at this text message:

In [20]:
```
print("MESSAGE 1 ORIGINAL: ", df.loc[660, 'msg'])
print("MESSAGE 1 CLEAN: ", df.loc[660, 'msg_clean'])
print("MESSAGE 1 TOKENIZED: ", [token for token in nlp(df.loc[660, 'msg_clean'])])
```

```
MESSAGE 1 ORIGINAL:  Under the sea, there lays a rock. In the rock, there is an envel
ope. In the envelope, there is a paper. On the paper, there are 3 words... '
MESSAGE 1 CLEAN:  sea lay rock rock envelope envelope paper paper 3 word
MESSAGE 1 TOKENIZED:  [sea, lay, rock, rock, envelope, envelope, paper, paper, 3, wor
d]
```

There are 10 tokens in the message (see `MESSAGE 1 TOKENIZED`) The `rock`, `envelope` and `paper` tokens all appear twice in the message, whereas the other tokens appear only once. This results in 7 unique tokens.

Using the BoW method, the features of this message would look like this:

In [22]:
```
{"sea": 1, "lay": 1, "rock": 2, "envelope": 2, "paper": 2, "3": 1, "word": 1}
```

Out[22]:
```
{'sea': 1, 'lay': 1, 'rock': 2, 'envelope': 2, 'paper': 2, '3': 1, 'word': 1}
```

Let's look at the other message:

In [23]:
```
print("MESSAGE 2 ORIGINAL: ", df.loc[1178, "msg"])
print("MESSAGE 2 CLEAN: ", df.loc[1178, "msg_clean"])
print("MESSAGE 2 TOKENIZED: ", [token for token in nlp(df.loc[1178, "msg_clean"])])
```

```
MESSAGE 2 ORIGINAL:  I'm outside islands, head towards hard rock and you'll run into
me
MESSAGE 2 CLEAN:  outside island head towards hard rock 'll run
MESSAGE 2 TOKENIZED:  [outside, island, head, towards, hard, rock, 'll, run]
```

Since each token only appears once in the message, the features could be represented as follows:

In [24]:
```
{"outside": 1, "island": 1, "head": 1, "towards": 1, "hard": 1, "rock": 1, "run": 1}
```

Out[24]:
```
{'outside': 1,
 'island': 1,
 'head': 1,
 'towards': 1,
 'hard': 1,
 'rock': 1,
 'run': 1}
```

Between these two messages there are 13 unique tokens in our sample corpus. These 13 tokens then serve as variables for the data set. In tabular form, the features then look like this:

message	sea	lay	rock	envelope	paper	3	word	outside	island	head	towa
Message 1	1	1	2	2	2	1	1	0	0	0	0
Message 2	0	0	1	0	0	0	0	1	1	1	1

To apply BoW vectorization, we use the *transformer* `CountVectorizer` from `sklearn.feature_extraction.text` - (link to the documentation). Use the following code cell to import `CountVectorizer`.

In [25]:
```python
from sklearn.feature_extraction.text import CountVectorizer
```

Now initialize `CountVectorizer()` and store it as a variable named `count_vectorizer`.

In [26]:
```python
count_vectorizer = CountVectorizer()
```

Now use the `my_transformer.fit_transform()` method to fit `count_vectorizer` to `features_train` and transform it. Usually this gives us a lot of features (all the unique words from all the messages). Most messages use only a small part of these features. So we end up with what's called a sparse matrix. This is a matrix that contains the value 0 a lot. To store them efficiently in the memory, you can use the datatype `.csr_matrix` (*compressed sparse row matrix*), which originally comes from the `scipy` module. `sklearn` can handle this like an array or a `DataFrame`. So we can use the transformed feature matrix as usual.

Store the transformed matrix in the new variable named `features_train_bow` and print its data type.

In [27]:
```python
features_train_bow = count_vectorizer.fit_transform(features_train)
features_train_bow
```

Out[27]:
```
<3900x6333 sparse matrix of type '<class 'numpy.int64'>'
	with 32648 stored elements in Compressed Sparse Row format>
```

You can see that the `features_train` training data set has been transformed into a `.csr_matrix`. Now let's look at how many unique words we have in the corpus, i.e. how many features `features_train_bow` contains. Use the `get_feature_names()` method of `count_vectorizer`, store the *output* in the variable `bow_features` and output its length.

In [30]:
```python
bow_features = count_vectorizer.get_feature_names()
len(bow_features)
```

Out[30]:
```
6333
```

You can see that we have ended up with `6333`. So we transformed the `features_train` training data set with only one column into a sparse matrix with `6333` columns. Let's take a closer look at this matrix. One disadvantage of `.csr_matrix` objects is that it is difficult to understand. But if we convert `features_train_bow` into a `DataFrame`, we can get around this problem. Run the following cell to do this.

In [31]:
```
bow_array = features_train_bow.toarray()
bow_vector = pd.DataFrame(bow_array, columns=bow_features)
bow_vector.head()
```

Out[31]:

	00	000	008704050406	0121	0125698789	02	0207	02072069400	02073162414	02085076972
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0

5 rows × 6333 columns

Each word or token is displayed in individual columns. To understand how BoW vectorized the tokens, let's look at the token `'call'`.

In the cell below, print the first 12 values of `bow_vector` in the `'call'` column.

In [39]:
```
mask = bow_vector.loc[:,'call'] != 0
bow_vector.loc[mask,'call']
```

Out[39]:
```
3       1
7       1
11      3
18      1
19      1
       ..
3829    1
3835    1
3845    1
3849    1
3893    1
Name: call, Length: 413, dtype: int64
```

The result shows three non-zero values in rows 3, 7 and 11.

Rows 3 and 7 have a value of `1`, i.e. the word `'call'` appears once in these messages.

In [40]:
```
# Printing the messages in rows 3 and 7
print("Message 3: ", features_train.iloc[3])
print("Message 7: ", features_train.iloc[7])
```

```
Message 3:  double mins 1000 txt orange tariff latest motorola sonyericsson nokia blu
etooth free call mobileupd8 08000839402 call2optout hf8
Message 7:  work please call
```

However, in row 11, the value for `'call'` is `3`, meaning the word appears 3 times. The word `'call'` does not appear in all other rows that have the value `0`. You are welcome to check this in the following cell.

```
In [41]: # Printing the messages in row 11
         print("Message 11: ", features_train.iloc[11])
```

```
Message 11:  santa call would little one like call santa xmas eve call 09077818151 bo
ok time calls1 50ppm last 3min 30 t&c www santacalling com
```

Congratulations: You have just learned how the *bag of words* vectorization method works and how to use it with `sklearn`. In the next section you will learn about another vectorization technique that goes beyond simple word counting.

call## The *term frequency - inverse document frequency* method

While the BoW method vectorizes words based on their frequency in the text, the Term Frequency-Inverse Document Frequency (**TF-IDF**) vectorization takes the *relevance* of a word into account.

To determine the relevance of a word, TF-IDF takes two things into account, as its name suggests:

1. How often a **word** appears in an **individual message**, based on the number of words in the same **message** (*term frequency*, the **TF** in **TF-IDF**).

\begin{equation*} \mathrm{TF}\left(\textbf{word}, \textbf{message} \right) = \frac{\mathrm{number\, of\, instances\, of\, a\, word\, \textbf{word} \,in\, a\, \textbf{message}}}{\mathrm{number\, of\, all\, words\, in\, the\, same\,\textbf{message}}} \end{equation*}

1. How many **messages** contain a **certain word**, in relation to the **size of the corpus** (*inverse document frequency*, the **IDF** in **TF-IDF**).

\begin{equation*} \mathrm{IDF}\left(\textbf{word}, \textbf{corpus}\right) = \ln\left(\frac{\mathrm{number \, of \,\textbf{messages}\, in\, the \, \textbf{corpus}}}{\mathrm{number\, of \, \textbf{messages}, \, that\, contain\, the \, \textbf{word}}}\right) \end{equation*}
This is a *natural logarithm*. The **TF-IDF** value of a **word** in a **message** in the **corpus** is defined as the product of **TF** and **IDF**.

\begin{equation*} \mathrm{TF\text{-}IDF} (\textbf{word},\textbf{message}, \textbf{corpus}) = \mathrm{TF}(\textbf{word},\textbf{message}) \,\cdot \, \mathrm{IDF}(\textbf{word}, \textbf{corpus}) \end{equation*}
The more **messages** the **word** appears in, the less valuable this **word** is for differentiating between text types. The **TF-IDF** of this kind of **word** would be small. An important **word** would be one that occurs very rarely in the entire **corpus**.

To demonstrate how TF-IDF works, let's take three messages from our original DataFrame `df` and use them to form a sample corpus. Execute the following cell to display the tokenized version of the messages:

In [42]:
```
# Tokenizing Message 1
message_1_tokenized = [token for token in nlp(df.loc[660, 'msg_clean'])]
print("MESSAGE 1 TOKENIZED: ", message_1_tokenized)

# Tokenizing Message 2
message_2_tokenized  = [token for token in nlp(df.loc[1178, 'msg_clean'])]
print("MESSAGE 2 TOKENIZED: ", message_2_tokenized)

# Tokenizing Message 3
message_3_tokenized  = [token for token in nlp(df.loc[621, 'msg_clean'])]
print("MESSAGE 3 TOKENIZED: ", message_3_tokenized)
```

```
MESSAGE 1 TOKENIZED:  [sea, lay, rock, rock, envelope, envelope, paper, paper, 3, wor
d]
MESSAGE 2 TOKENIZED:  [outside, island, head, towards, hard, rock, 'll, run]
MESSAGE 3 TOKENIZED:  [good, word, word, may, leave, u, dismay, many, time]
```

Our corpus consists of the top three messages. Let's start by determining the TF value of `"word"` and `"rock"` in message 1. Run the following cell.

In [43]:
```
# Length of tokenized Message 1
message_1_tokenized_len = len(message_1_tokenized)
print("NUMBER OF WORDS IN MESSAGE 1: ", message_1_tokenized_len)

# Calculating TF of `word` which appears once in the first message
message_1_tf_word = 1 / message_1_tokenized_len

# Calculating TF of `rock` which appears twice in the first message
message_1_tf_rock = 2 / message_1_tokenized_len

print("TF OF 'word' IN MESSAGE 1: ", message_1_tf_word)
print("TF OF 'rock' IN MESSAGE 1: ", message_1_tf_rock)
```

```
NUMBER OF WORDS IN MESSAGE 1:  10
TF OF 'word' IN MESSAGE 1:  0.1
TF OF 'rock' IN MESSAGE 1:  0.2
```

`"word"` appears once and `"rock"` appears twice. This means that the latter receives a higher TF value.

In message 2, both words appear only once. Let's calculate the TF values of `"rock"` and `"run"` in message 2. Run the following cell:

In [44]:
```
# Length of tokenized Message 2
message_2_tokenized_len = len(message_2_tokenized)
print("NUMBER OF WORDS IN MESSAGE 2: ", message_2_tokenized_len)

# Calculating TF of `run` which appears once in the second message
message_2_tf_run = 1 / message_2_tokenized_len

# Calculating TF of `rock` which appears once in the second message
message_2_tf_rock = 1 / message_2_tokenized_len
```

```
print("TF OF 'run' IN MESSAGE 2: ", message_2_tf_run)
print("TF OF 'rock' IN MESSAGE 2: ", message_2_tf_rock)
```

```
NUMBER OF WORDS IN MESSAGE 2:  8
TF OF 'run' IN MESSAGE 2:  0.125
TF OF 'rock' IN MESSAGE 2:  0.125
```

Now let us take the third message and calculate the TF value of `"word"`, which appears twice in it.

In [45]:
```
# Length of tokenized Message 3
message_3_tokenized_len = len(message_3_tokenized)
print("NUMBER OF WORDS IN MESSAGE 3: ", message_3_tokenized_len)

# Calculating TF of `word` which appears twice in the third message
message_3_tf_word = 2 / message_3_tokenized_len

print("TF OF 'word' IN MESSAGE 3: ", message_3_tf_word)
```

```
NUMBER OF WORDS IN MESSAGE 3:  9
TF OF 'word' IN MESSAGE 3:  0.2222222222222222
```

Let let's look at the IDF values. Execute the following cell to calculate it for `"rock"` (appears in two messages), `"word"` (appears in two messages) and `"run"` (appears in one message) with a corpus size of three messages:

In [46]:
```
import math
# There are 3 messages in our corpus

# 'rock' appears in messages 1 and 2
idf_rock = math.log(3 / 2)

# 'word' appears in messages 1 and 3
idf_word = math.log(3 / 2)

# 'run' appears in message 2
idf_run = math.log(3 / 1)

print("IDF of 'rock': ", idf_rock)
print("IDF of 'word': ", idf_word)
print("IDF of 'run': ", idf_run)
```

```
IDF of 'rock':  0.4054651081081644
IDF of 'word':  0.4054651081081644
IDF of 'run':  1.0986122886681098
```

In contrast to the TF value, which increases as the frequency of a word increases, words receive a higher IDF value the rarer they are (in the example above, `"run"` is the rarest word).

Now let's take a closer look at the TF-IDF values.

Here are the TF-IDF values of `"rock"`, which appears three times in two messages:

In [47]:
```
print("TF-IDF of 'rock' in Message 1: ", message_1_tf_rock * idf_rock)
print("TF-IDF of 'rock' in Message 2: ", message_2_tf_rock * idf_rock)
```

```
TF-IDF of 'rock' in Message 1:  0.08109302162163289
TF-IDF of 'rock' in Message 2:  0.05068313851352055
```

Note that TF-IDF of `"rock"` in message 1 is slightly larger than in message 2. So `"rock"` is more important for message 1.

Here are the TF-IDF values of `"word"`, which also appears three times in two messages are:

In [48]:
```
print("TF-IDF of 'word' in Message 1: ", message_1_tf_word * idf_word)
print("TF-IDF of 'word' in Message 3: ", message_3_tf_word * idf_word)
```

```
TF-IDF of 'word' in Message 1:  0.04054651081081644
TF-IDF of 'word' in Message 3:  0.09010335735736986
```

In message 3, the TF-IDF value of `"word"` is larger and therefore more important for this message.

Finally, let's look at the TF-IDF value of `"run"` in message 2:

In [49]:
```
print("TF-IDF of 'run' in Message 2: ", message_2_tf_run * idf_run)
```

```
TF-IDF of 'run' in Message 2:  0.13732653608351372
```

Since `"run"` only appears in one message, its TF-IDF value is higher than `"word"` and `"rock"`, which appear in two of three messages.

We can summarize the TF-IDF values in a table.

message	rock	word	run
Message 1	0,081	0,041	0
Message 2	0,051	0	0,137
Message 3	0	0,090	0

`sklearn` offers another transformer which does the TF-IDF vectorization for us. It's called `TfidfVectorizer` and can also be found in `sklearn.feature_extraction.text` - (link to documentation). Import it in the following cell.

In [50]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

Initialize `TfidfVectorizer` and then use the `my_transformer.fit_transform()` method to fit the instance to `features_train` and transform it. Store the result in the variable `tfidf_vectorizer` and `features_train_tfidf`.

In [52]:
```
tfidf_vectorizer = TfidfVectorizer()
features_train_tfidf = tfidf_vectorizer.fit_transform(features_train)
```

How many unique tokens are there? Use the `my_transformer.get_feature_names()` method of `tfidf_vectorizer` to get the names of the features and print their number. Store these distances in the variable `tfidf_features`.

In [58]:
```
tfidf_features = tfidf_vectorizer.get_feature_names()
len(tfidf_features)
```
Out[58]:
```
6333
```

As expected, we got back `6333` features just like with `CountVectorizer`. Since `features_train_tfidf` is a `csr_matrix`, we'll convert it to a `DataFrame` to understand it better. Run the following cell to do this.

In [59]:
```
tfidf_vector = pd.DataFrame(features_train_tfidf.toarray(), columns = tfidf_features)
tfidf_vector.head()
```
Out[59]:

	00	000	008704050406	0121	0125698789	02	0207	02072069400	02073162414	02085076972
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 6333 columns

The `fidf_vectorizer.idf_` attribute specifies the IDF values of each token. Save the resulting array as `idf_values`. Then print the maximum and minimum IDF values.

In [63]:
```
idf_values = tfidf_vectorizer.idf_
print(min(idf_values))
print(max(idf_values))
```
```
3.2407096892759584
8.575841028946712
```

To compare the results of BoW and TF-IDF, let's take another look at the results of the *token* `'call'`:

In [64]:
```
print("IDF VALUE OF CALL: ", idf_values[tfidf_features.index("call")])
```
```
IDF VALUE OF CALL:  3.243122235681342
```

Comparing the IDF value of `'call'` with the minimum and maximum value, it appears that 'call' is a fairly common word throughout the corpus, as the value is close to the minimum.

Now print the first 12 rows of `bow_vector` and `tfidf_vector` with the `"call"` column.

In [66]:
```
print(bow_vector.loc[:12, 'call'])
print(tfidf_vector .loc[:12, 'call'])
```

```
0     0
1     0
2     0
3     1
4     0
5     0
6     0
7     1
8     0
9     0
10    0
11    3
12    0
Name: call, dtype: int64
0     0.000000
1     0.000000
2     0.000000
3     0.119834
4     0.000000
5     0.000000
6     0.000000
7     0.437615
8     0.000000
9     0.000000
10    0.000000
11    0.292693
12    0.000000
Name: call, dtype: float64
```

Note that `'call'` in rows 3 and 7 has the same value of `1` with BoW vectorization. However, for TF-IDF, the value in row 7 is much higher (`0.44`) than in row 3 (`0.12`). Why would that be? Run the following cell to see these messages again.

In [67]:
```
# Messages
print("Message 3: ", features_train.iloc[3])
print("Message 7: ", features_train.iloc[7])
```

```
Message 3:  double mins 1000 txt orange tariff latest motorola sonyericsson nokia blu
etooth free call mobileupd8 08000839402 call2optout hf8
Message 7:  work please call
```

TF-IDF considers the fact that message 7 only has three tokens. This means that in relation to message 3, the word `"call"` is more important for understanding the meaning of message 7 than in message 3. This example shows how TF-IDF captures subtle nuances not found in BoW.

In [ ]:
```
print("Message 11: ", features_train.iloc[11])
```

For message 11, the *token* `"call"` appeared three times, so its BoW value is greater than for message 3 and 7. However, the TF-IDF is relatively low at 0.29. The frequency of the word occurring was probably compensated for by the total number of tokens in the message and the relatively low IDF value of `"call"`.

Congratulations: You have just learned how to use TF-IDF vectorization with the help of `sklearn`. Your contacts in the company are delighted that you have converted the text so that you can use it with machine learning algorithms. They are already curious to see how well your model will detect the spam messages.

Remember:

- The BoW method (`CountVectorizer`) vectorizes words based on their frequencies in the text.
- TF-IDF vectorization (`TfidfVectorizer`) takes into account the *relevance* of a word.

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.