Text Data Processing

Contents

8.1. Introduction

8.1.1 What is Text data

Text data usually consists of documents that can represent words, sentences or even paragraphs of free-flowing text. The inherent unstructured (no neatly formatted data columns!) and noisy nature of textual data makes it harder for machine learning methods to directly work on raw text data. Text analysis is an umbrella term encompassing AI-empowered techniques that help derive meaningful information from unstructured data. These insights, in turn, help make informed, data-backed decisions, enhance productivity, and improve business intelligence.

Text analysis can:

- Help analyze customer preferences, trends, and needs, assisting you in developing better products and features.
- Help study a substantial amount of data in real-time – without occupying your team’s time. Since text analysis with AI reduces manual work, productivity soars.
- Reduce the scope of error - by encompassing Machine Learning and Natural Language Processing (NLP) to unify criteria of analysis.

8.1.2 Understanding Text Data

We all have a fair idea of what textual data comprises. In general you can always have text data in the form of structured data attributes, but usually those fall under the umbrella of structured, categorical data. In this scenario, we are talking about free flowing text in the form of words, phrases, sentences and entire documents. Essentially, we do have some syntactic structure like words make phrases, phrases make sentences which in turn make paragraphs. However, there is no inherent structure to text documents because you can have a wide variety of words which can vary across documents and each sentence will also be of variable length as compared to a fixed number of data dimensions in structured datasets.

8.1.3 Application of Text Data

Common use cases for Text Analysis

- **Healthcare:** The industry uses text analysis to find patterns in doctors’ reports, identifying patterns in patient data. You can also use it to detect disease outbreaks by discovering cases in social media data.

- **Research:** Researchers use text analysis with AI to explore pre-existing literature to identify trends and patterns - or categorizing research survey answers by topic or sentiment.
- **Product development:** By analyzing boatloads of customer reviews and trends, text analysis helps determine in-demand features. By analyzing customer reviews on Amazon, a young analysts team, for instance, studied the price customers were happy to pay for a new market they were tapping into.
- **Customer service and experience:** By automatically studying various data such as critical tickets, call notes, surveys, and more, businesses can identify urgent requests to respond to and discover sentiment around their product/service.

8.1.4 Text Analysis vs. Text Mining vs. Text Analytics

Text Analysis and Text mining: Both the terms are commonly used interchangeably - and rightfully so. The roots of text analysis lie in social sciences, while text mining is delved from computer science. In today's context, however, they both refer to obtaining data through various statistical techniques.

Text mining uses techniques like Machine Learning and NLP to pull information about sentiment, urgency, emotion, or topical categories and context out of structured data - essentially to understand human language. Text mining or text analysis techniques could therefore identify customers' sentiment towards your product or brand based on survey responses or feedback forms.

Text mining processes typically include speech tagging, syntactic parsing, named entity recognition, but also more basic techniques for acquiring and processing data - e.g. web scraping and crawling in order to make use of dictionaries and other lexical resources and for processing texts and relating words.

In a nutshell - text analysis is used for qualitative insights - detecting sentiment in language, or topics and context in any free-form text.

Text Analysis vs Text Analytics: The other side of this coin is text analytics that focuses on quantitative insights. Text analytics draws valuable, recurrent, and emerging patterns, themes, and trends from text-based data - e.g. identifying patterns from data gathered over a year to determine annual trends. With text analysis, we can identify the sentiment of our customer survey responses. If we wanted to additionally focus on metrics like how many surveys were completed in which timeframe or location, we would opt for a text analytics tool that creates graphs, tables, or reports.

Choosing the correct text analytics technique depends on the dataset available. In most cases, you’ll need to use a combination of two techniques or more to get actionable insights.

8.1.5 Text analysis VS natural language processing (NLP)

Natural language processing is actually a subset of the broader text analysis field, which is why the difference between the two can often be hard to comprehend. Let’s start with the definitions of text analysis and natural language processing. Text analysis is about examining large collections of text to generate new and relevant insights.

Natural language processing (NLP), or more specifically, natural language understanding (NLU), helps machines “read”, “understand” and replicate human speech. In the process of text analysis, various analysis methods are used to derive insights, and natural language processing is one of them. NLP is actually an interdisciplinary field between text analysis, computational linguistics, AI and machine learning.

The key difference between text analysis and NLP lies in the goals of each field. Text analysis aims to derive quality insights from solely the text or words itself. Semantics in the text is not considered. It answers questions like frequency of words, length of sentence, and presence or absence of words.

On the other hand, NLP aims to understand the linguistic use and context behind the text. Here, grammatical structures and semantics are analysed. It answers questions like the intention behind a sentence, people’s linguistic habits and even classify which of your emails should go into the Primary, Social, Promotions or Updates tabs. The two are often used together to provide both a numerical and contextual understanding of human communications.

So, how are these used in business? The next section will cover use cases and applications of text analysis.

8.1.6 Challenges of Text Analysis

Any employee who has ever had to review customer support tickets or the company's info@ mailbox will instantly know the benefits of using a text analysis software:

- It is scalable - no matter the volume of incoming text, the software can handle the variance.
- It is consistent - humans make mistakes - that's natural. Or in cases where you work in a team, opinions and criteria for decision-making can differ.
- It analyses text in real-time - analyzing incoming emails at scale can take minutes instead of days and weeks.

There are however some challenges that should not be overlooked:

- The ambiguity of human language - An *"apple"* can refer to a fruit or the company, and context matters a great deal in distinguishing between the two. One way to mitigate this is to add NLP capabilities to your pipeline.

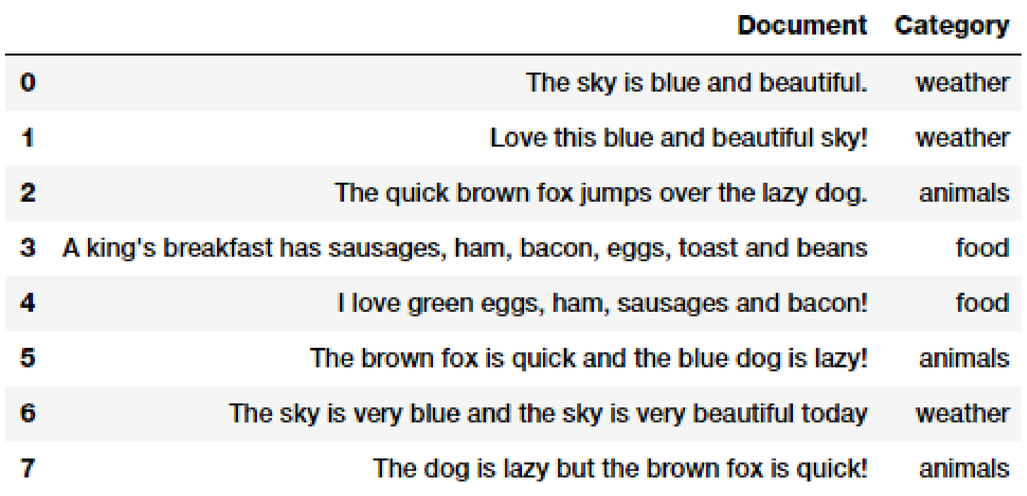
- Multi-lingual scenarios - most text mining algorithms operate in a specific language making processing multilingual documents inefficient.
- Usability - Text analysis tools are more often than not designed for trained knowledge workers and are typically too complex for the average business professional.

8.2 Feature Engineering Strategies

Let’s look at some popular and effective strategies for handling text data and extracting meaningful features from the same which can be used in downstream machine learning systems. Let’s now take a sample corpus of documents. A corpus is typically a collection of text documents usually belonging to one or more subjects. We will show some basic python code as well and We’ll start by loading up some basic dependencies and settings.

```
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib.pyplot as pltpd.options.display.max_colwidth = 200
#matplotlib inline
```

```
corpus = ['The sky is blue and beautiful.',
          'Love this blue and beautiful sky!',
          'The quick brown fox jumps over the lazy dog.',
          "A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
          'I love green eggs, ham, sausages and bacon!',
          'The brown fox is quick and the blue dog is lazy!',
          'The sky is very blue and the sky is very beautiful today',
          'The dog is lazy but the brown fox is quick!'
]
labels = ['weather', 'weather', 'animals', 'food', 'food', 'animals', 'weather', 'animals']

corpus = np.array(corpus)
corpus_df = pd.DataFrame({'Document': corpus,
                          'Category': labels})
corpus_df = corpus_df[['Document', 'Category']]


corpus_df
```



	Document	Category
0	The sky is blue and beautiful.	weather
1	Love this blue and beautiful sky!	weather
2	The quick brown fox jumps over the lazy dog.	animals
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food
4	I love green eggs, ham, sausages and bacon!	food
5	The brown fox is quick and the blue dog is lazy!	animals
6	The sky is very blue and the sky is very beautiful today	weather
7	The dog is lazy but the brown fox is quick!	animals

You can see that we have taken a few sample text documents belonging to different categories for our toy corpus. Before we focus on feature engineering, as always, we need to do some data pre-processing or wrangling to remove unnecessary characters, symbols and tokens.

8.3 Text pre-processing

There can be multiple ways of cleaning and pre-processing textual data. In the following points, we highlight some of the most important ones which are used heavily in Natural Language Processing (NLP) pipelines.

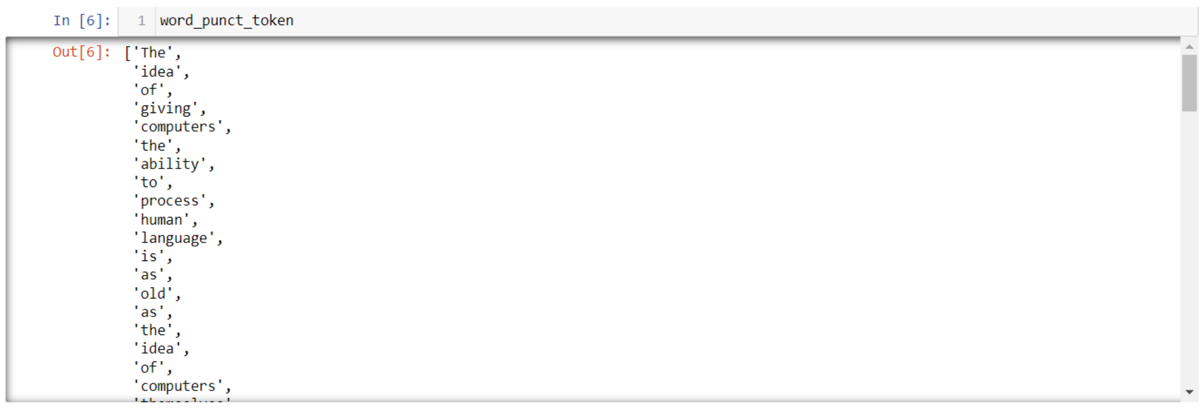
Consider the following example for text pre-processing:

```
#Sample Text

text = """ The idea of giving computers the ability to process human language is as old as the idea of computers themselves. This book is about the implementation and implications of that exciting idea. We introduce a vibrant interdisciplinary field with many names corresponding to its many facets, names like speech and language processing, human language technology, natural language processing, computational linguistics, and speech recognition and synthesis. The goal of this new field is to get computers to perform useful tasks involving human language, tasks like enabling human-machine communication, improving human-human communication, or simply doing useful processing of text or speech. """
```

8.3.1 Tokenization

Using tokenizer to separate the sentences into a list of single words (tokens).

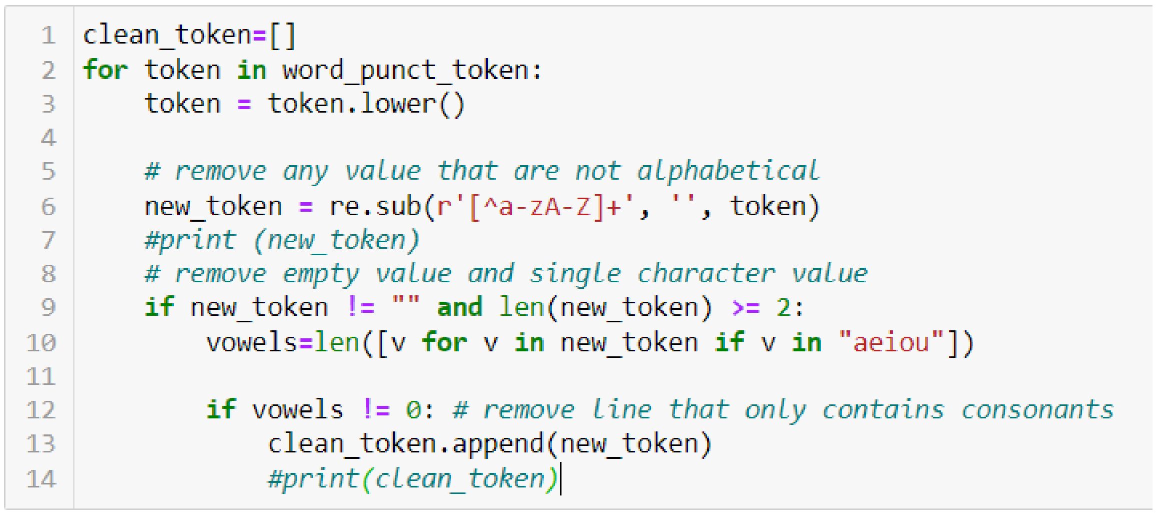
```
word_punct_token = WordPunctTokenizer().tokenize(text)
```



```
In [6]:   word_punct_token

Out[6]: ['The',
         'idea',
         'of',
         'giving',
         'computers',
         'the',
         'ability',
         'to',
         'process',
         'human',
         'language',
         'is',
         'as',
         'old',
         'as',
         'the',
         'idea',
         'of',
         'computers',
```

8.3.2 Normalization

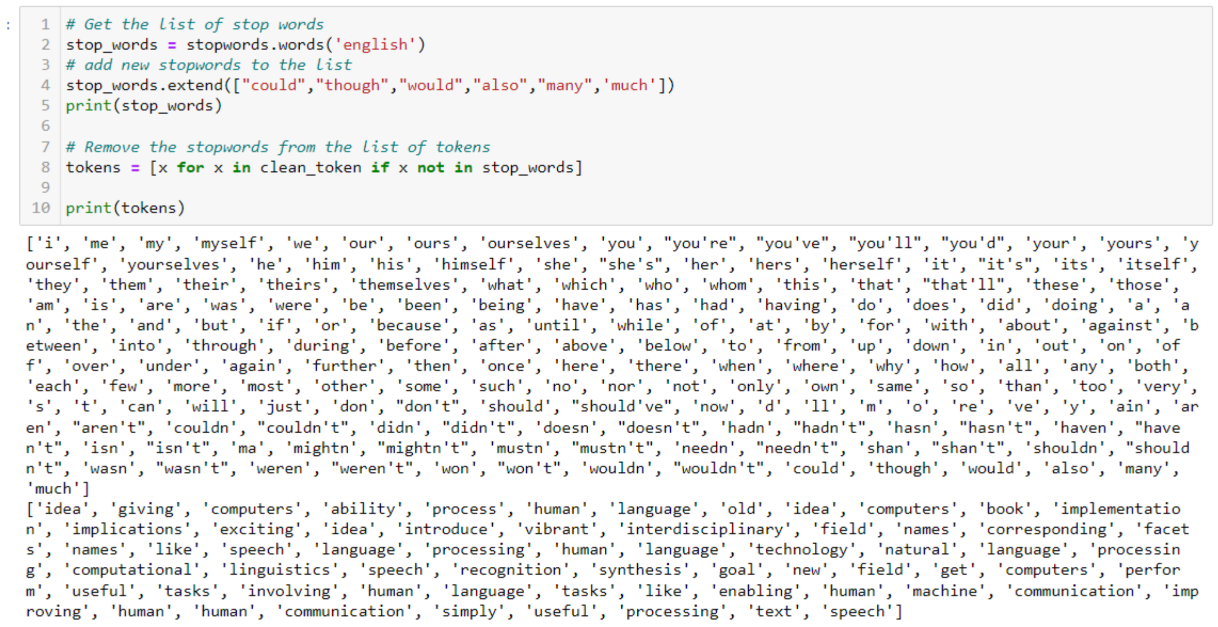
The script below removed the tokens which are not a word, for example, the symbols and numbers, also tokens that only contain less than two letters or contain only consonants. This script might not be useful in this example, but it's pretty useful when you are dealing with a large set of text data, it helps to clean up much noise. I always like to include it whenever I was processing text data.



```python
clean_token=[]
for token in word_punct_token:
    token = token.lower()

    # remove any value that are not alphabetical
    new_token = re.sub(r'[^a-zA-Z]+', '', token)
    #print (new_token)
    # remove empty value and single character value
    if new_token != "" and len(new_token) >= 2:
        vowels=len([v for v in new_token if v in "aeiou"])

        if vowels != 0: # remove line that only contains consonants
            clean_token.append(new_token)
            #print(clean_token)
```

8.3.3 Removing Stopwords

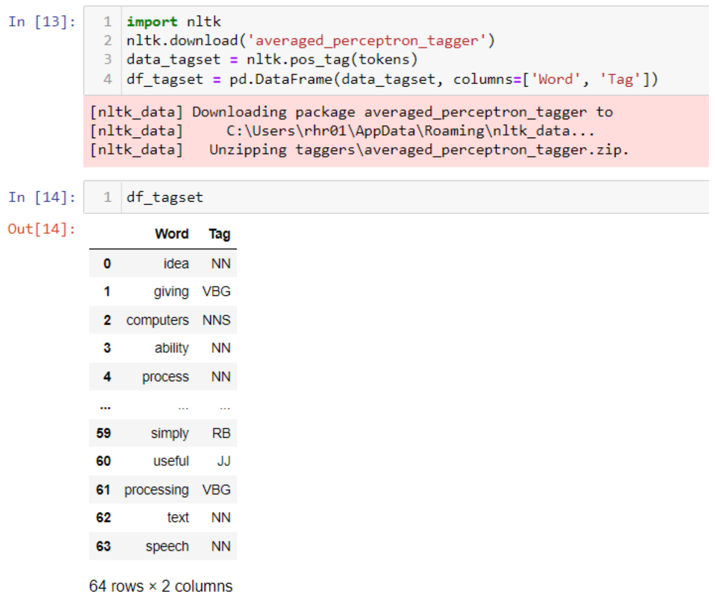
Stopwords referring to the word which does not carry much insight, such as preposition. NLTK and spaCy have a different amount of stopwords in the library, but both NLTK and spaCy allowed us to add in any word we feel necessary. For example, when we dealing with email, we may add in gmail, com, outlook as stopwords.


```python
# Get the list of stop words
stop_words = stopwords.words('english')
# add new stopwords to the list
stop_words.extend(["could","though","would","also","many",'much'])
print(stop_words)

# Remove the stopwords from the list of tokens
tokens = [x for x in clean_token if x not in stop_words]

print(tokens)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'y
ourself', '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', 'this', 'that', "that'll", 'these', 'those',
'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'a
n', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'b
etween', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'of
f', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most', 'other', 'some', 'such', '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', 'ar
en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have
n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't", 'could', 'though', 'would', 'also', 'many',
'much']
['idea', 'giving', 'computers', 'ability', 'process', 'human', 'language', 'old', 'idea', 'computers', 'book', 'implementatio
n', 'implications', 'exciting', 'idea', 'introduce', 'vibrant', 'interdisciplinary', 'field', 'names', 'corresponding', 'facet
s', 'names', 'like', 'speech', 'language', 'processing', 'human', 'language', 'technology', 'natural', 'language', 'processin
g', 'computational', 'linguistics', 'speech', 'recognition', 'synthesis', 'goal', 'new', 'field', 'get', 'computers', 'perfor
m', 'useful', 'tasks', 'involving', 'human', 'language', 'tasks', 'like', 'enabling', 'human', 'machine', 'communication', 'imp
roving', 'human', 'human', 'communication', 'simply', 'useful', 'processing', 'text', 'speech']
```

8.3.4 Part-of-Speech Tagging (POS Tag)

This process referring to tag the word with their part-of-speech position, for example, verbs, adjectives and nouns. nltk.pos_tag module return results as tuples, to ease the task afterwards, normally I will transform them into a DataFrame. The POS Tag is the task perform straight after tokenization, which is a smart move when you know you only need a particular part of speech like adjectives and nouns.


In [13]:
```python
import nltk
nltk.download('averaged_perceptron_tagger')
data_tagset = nltk.pos_tag(tokens)
df_tagset = pd.DataFrame(data_tagset, columns=['Word', 'Tag'])
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\rhr01\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping taggers\averaged_perceptron_tagger.zip.
```

In [14]:
```python
df_tagset
```

Out[14]:

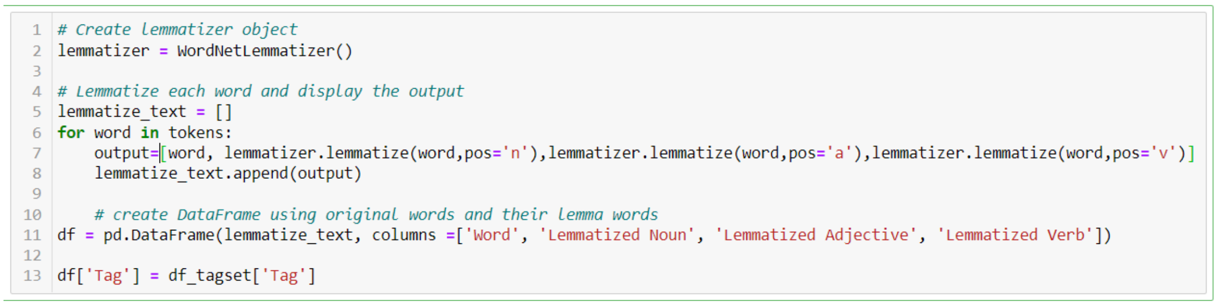
	Word	Tag
0	idea	NN
1	giving	VBG
2	computers	NNS
3	ability	NN
4	process	NN
...	...	...
59	simply	RB
60	useful	JJ
61	processing	VBG
62	text	NN
63	speech	NN

64 rows × 2 columns

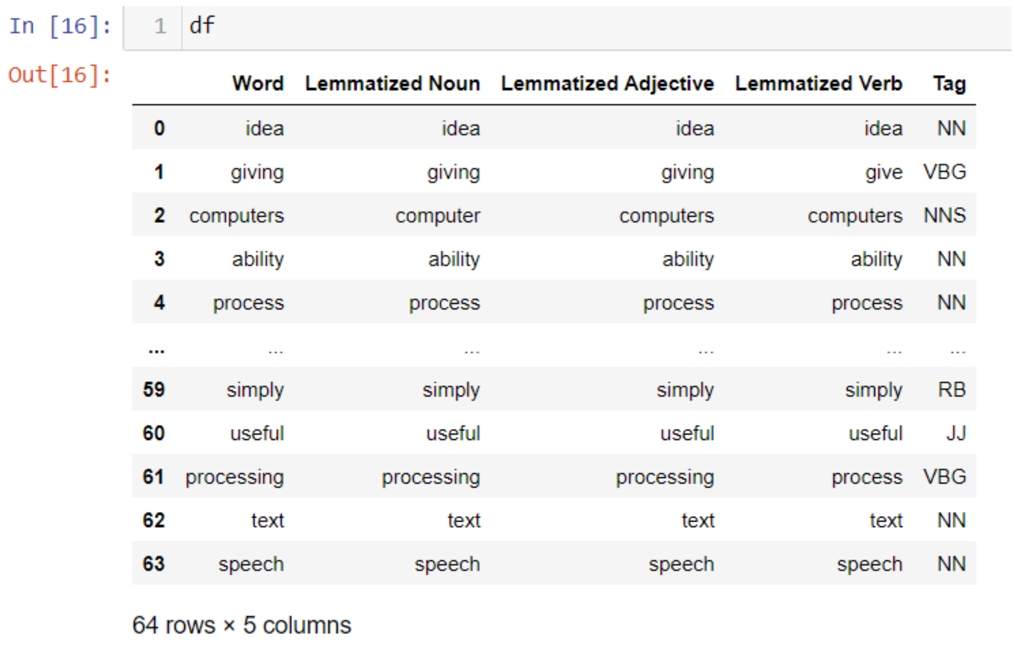
8.3.5 Lemmatization

Lemmatizing and stemming both help to reduce the dimension of the vocabulary by return the words to their root form (lemmatizing) or remove all the suffix, affix, prefix and so on (stemming). Stemming is nice for reducing the dimension of vocabulary, but most of the time the word become meaningless as stemming only chopped off the suffix but not returning the words to their base form. For example, houses will become hous after stemming, which completely lose its meaning. Hence, lemmatizing is more preferable for text analytics.

The following script is used to obtain the root form for the nouns, adjectives and verbs.



```python
# Create lemmatizer object
lemmatizer = WordNetLemmatizer()

# Lemmatize each word and display the output
lemmatize_text = []
for word in tokens:
    output=[word, lemmatizer.lemmatize(word,pos='n'),lemmatizer.lemmatize(word,pos='a'),lemmatizer.lemmatize(word,pos='v')]
    lemmatize_text.append(output)

    # create DataFrame using original words and their lemma words
df = pd.DataFrame(lemmatize_text, columns =['Word', 'Lemmatized Noun', 'Lemmatized Adjective', 'Lemmatized Verb'])

df['Tag'] = df_tagset['Tag']
```



```
In [16]: df
```

Out[16]:

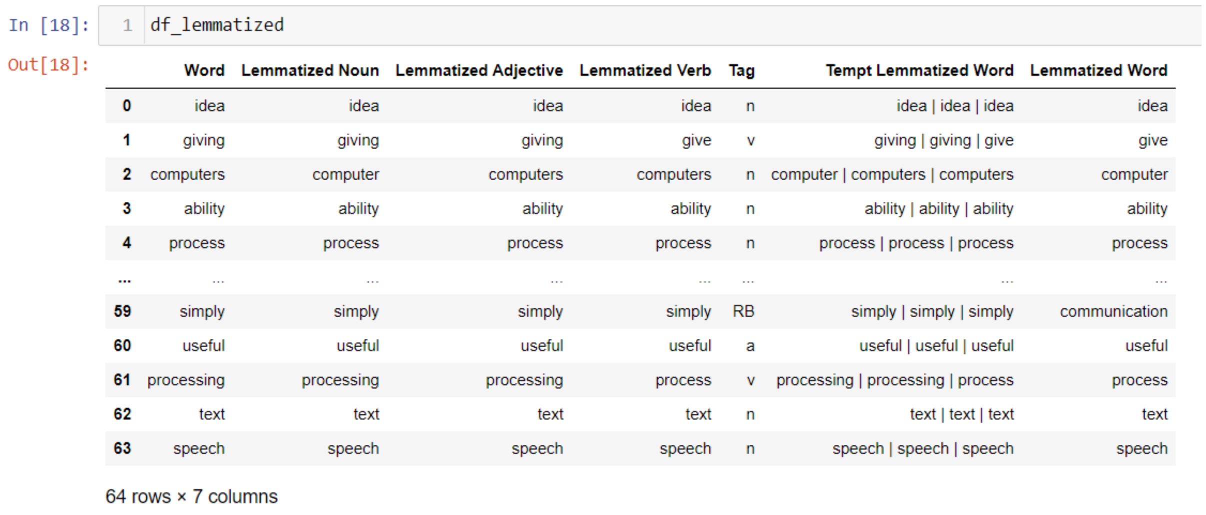
	Word	Lemmatized Noun	Lemmatized Adjective	Lemmatized Verb	Tag
0	idea	idea	idea	idea	NN
1	giving	giving	giving	give	VBG
2	computers	computer	computers	computers	NNS
3	ability	ability	ability	ability	NN
4	process	process	process	process	NN
...	...	...	...	...	...
59	simply	simply	simply	simply	RB
60	useful	useful	useful	useful	JJ
61	processing	processing	processing	process	VBG
62	text	text	text	text	NN
63	speech	speech	speech	speech	NN

64 rows × 5 columns

The reason for lemmatizing the adjectives, nouns and verbs separately is to improve the accuracy of the lemmatizer.

```
# replace with single character for simplifying

df = df.replace(['NN','NNS','NNP','NNPS'],'n')

df = df.replace(['JJ','JJR','JJS'],'a')

df = df.replace(['VBG','VBP','VB','VBD','VBN','VBZ'],'v')

'''

define a function where take the lemmatized word when tagset is noun, and take lemmatized adjectives when tagset is adjective

'''

df_lemmatized = df.copy()

df_lemmatized['Tempt Lemmatized Word']=df_lemmatized['Lemmatized Noun'] + ' | ' + df_lemmatized['Lemmatized Adjective']+ ' | ' + df_lemmatized['Lemmatized Verb']

df_lemmatized.head(5)
```

```
lemma_word = df_lemmatized['Tempt Lemmatized Word']
tag = df_lemmatized['Tag']
i = 0
new_word = []
while i<len(tag):
    words = lemma_word[i].split('|')
    if tag[i] == 'n':
        word = words[0]
    elif tag[i] == 'a':
        word = words[1]
    elif tag[i] == 'v':
        word = words[2]
    new_word.append(word)
    i += 1
df_lemmatized['Lemmatized Word']=new_word
```



```
In [18]: df_lemmatized
Out[18]:
```

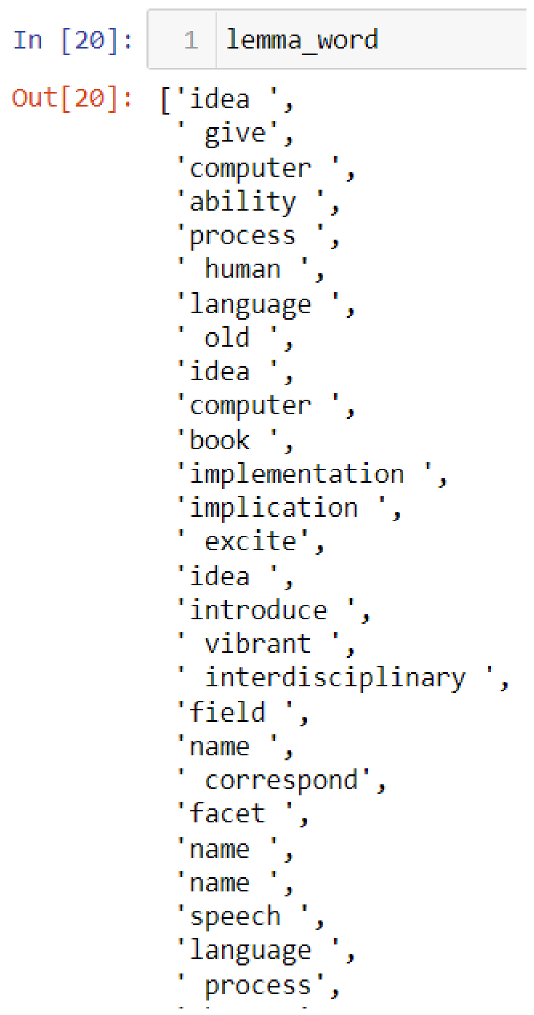
	Word	Lemmatized Noun	Lemmatized Adjective	Lemmatized Verb	Tag	Tempt Lemmatized Word	Lemmatized Word
0	idea	idea	idea	idea	n	idea | idea | idea	idea
1	giving	giving	giving	give	v	giving | giving | give	give
2	computers	computer	computers	computers	n	computer | computers | computers	computer
3	ability	ability	ability	ability	n	ability | ability | ability	ability
4	process	process	process	process	n	process | process | process	process
...	...	...	...	...	...	...	...
59	simply	simply	simply	simply	RB	simply | simply | simply	communication
60	useful	useful	useful	useful	a	useful | useful | useful	useful
61	processing	processing	processing	process	v	processing | processing | process	process
62	text	text	text	text	n	text | text | text	text
63	speech	speech	speech	speech	n	speech | speech | speech	speech

64 rows × 7 columns

The script above is to assign the correct lemmatized word to the original word according to their POS Tag.

8.3.6 Obtain the Cleaned Tokens

```
# calculate frequency distribution of the tokens
lemma_word = [str(x) for x in df_lemmatized['Lemmatized Word']]
```



```
In [20]:  lemma_word

Out[20]: ['idea ',
           '  give',
           'computer ',
           'ability ',
           'process ',
           '  human ',
           'language ',
           '  old ',
           'idea ',
           'computer ',
           'book ',
           'implementation ',
           'implication ',
           '  excite',
           'idea ',
           'introduce ',
           '  vibrant ',
           '  interdisciplinary ',
           'field ',
           'name ',
           '  correspond',
           'facet ',
           'name ',
           'name ',
           'speech ',
           'language ',
           '  process',
```

After tokenized and normalized the text, now we obtained a list of clean tokens, which ready to be plug-in into WordCloud or other text analytics models.

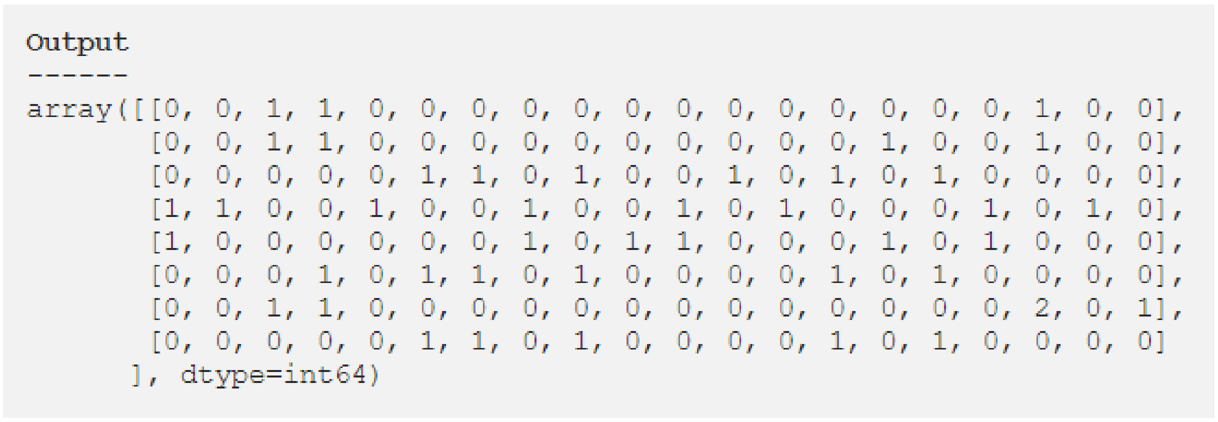
Note: for wordcloud, please see lecture slide and python example.

8.4 Vector Space Representation Model

8.4.1 Bag of Words Model

This is perhaps the most simple vector space representational model for unstructured text. A vector space model is simply a mathematical model to represent unstructured text (or any other data) as numeric vectors, such that each dimension of the vector is a specific feature\attribute. The bag of words model represents each text document as a numeric vector where each dimension is a specific word from the corpus and the value could be its frequency in the document, occurrence (denoted by 1 or 0) or even weighted values. The model's name is such because each document is represented literally as a 'bag' of its own words, disregarding word orders, sequences and grammar. You can see an example that a document have been converted into numeric vectors such that each document is represented by one vector (row) in the above feature matrix.

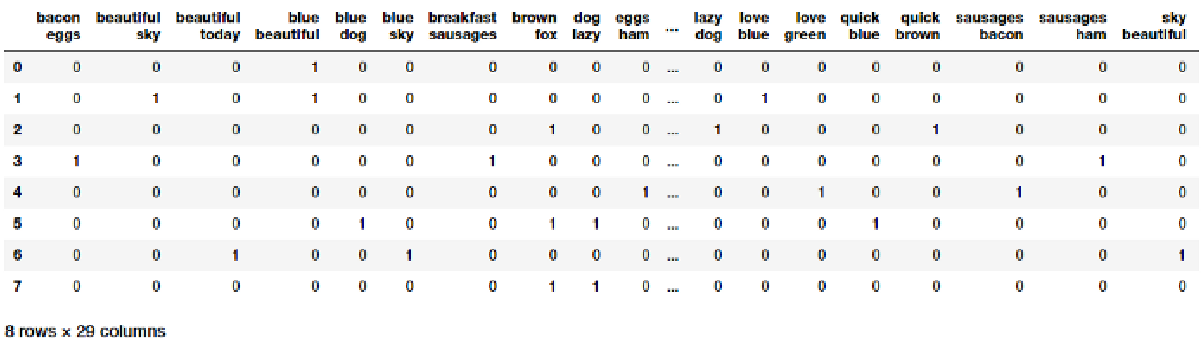
```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(min_df=0., max_df=1.)
cv_matrix = cv.fit_transform(norm_corpus)
cv_matrix = cv_matrix.toarray()
cv_matrix
```



```
Output
------
array([[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0],
       [1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1],
       [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0]
     ], dtype=int64)
```

The figure below should make things more clearer! You can clearly see that each column or dimension in the feature vectors represents a word from the corpus and each row represents one of our documents. The value in any cell, represents the number of times that word (represented by column) occurs in the specific document (represented by row). Hence if a corpus of documents consists of *N* unique words across all the documents, we would have an *N-dimensional* vector for each of the documents.

```
# get all unique words in the corpus
        vocab = cv.get_feature_names()
# show document feature vectors
pd.DataFrame(cv_matrix, columns=vocab)
```



	bacon eggs	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	breakfast sausages	brown fox	dog lazy	eggs ham	...	lazy dog	love blue	love green	quick blue	quick brown	sausages bacon	sausages ham	sky beautiful
0	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	...	1	0	0	0	1	0	0	0
3	1	0	0	0	0	0	1	0	0	0	...	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	1	...	0	0	1	0	0	1	0	0
5	0	0	0	0	1	0	0	1	1	0	...	0	0	0	1	0	0	0	0
6	0	0	1	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1	1	0	...	0	0	0	0	0	0	0	0

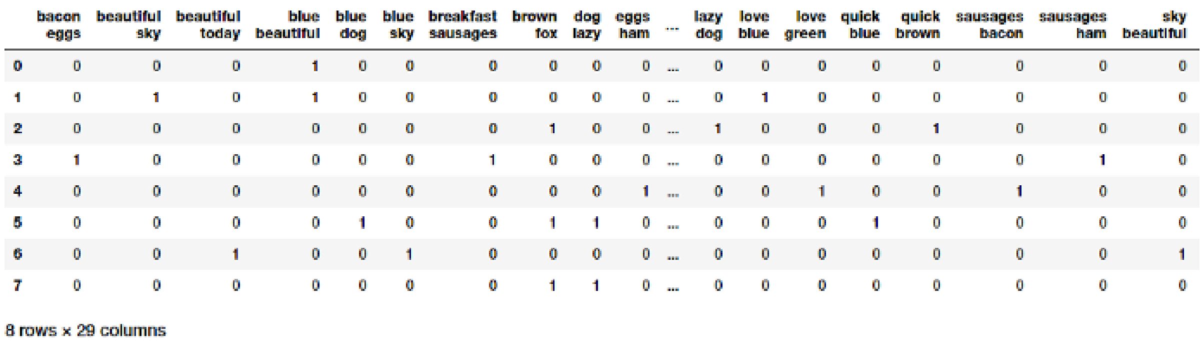
8 rows × 29 columns

This gives us feature vectors for our documents, where each feature consists of a bi-gram representing a sequence of two words and values represent how many times the bi-gram was present for our documents.

8.4.2 Bag of N-Grams Model

A word is just a single token, often known as a unigram or 1-gram. We already know that the Bag of Words model doesn't consider order of words. But what if we also wanted to take into account phrases or collection of words which occur in a sequence? N-grams help us achieve that. An N-gram is basically a collection of word tokens from a text document such that these tokens are contiguous and occur in a sequence. Bi-grams indicate n-grams of order 2 (two words), Tri-grams indicate n-grams of order 3 (three words), and so on. The Bag of N-Grams model is hence just an extension of the Bag of Words model so we can also leverage N-gram based features. The following example depicts bi-gram based features in each document feature vector.

```
# you can set the n-gram range to 1,2 to get unigrams as well as bigrams

bv = CountVectorizer(ngram_range=(2,2))
bv_matrix = bv.fit_transform(norm_corpus)
bv_matrix = bv_matrix.toarray()
vocab = bv.get_feature_names()
pd.DataFrame(bv_matrix, columns=vocab)
```



	bacon eggs	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	breakfast sausages	brown fox	dog lazy	eggs ham	...	lazy dog	love blue	love green	quick blue	quick brown	sausages bacon	sausages ham	sky beautiful
0	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	...	1	0	0	0	1	0	0	0
3	1	0	0	0	0	0	1	0	0	0	...	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	1	...	0	0	1	0	0	1	0	0
5	0	0	0	0	1	0	0	1	1	0	...	0	0	0	1	0	0	0	0
6	0	0	1	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1	1	0	...	0	0	0	0	0	0	0	0

8 rows × 29 columns

This gives us feature vectors for our documents, where each feature consists of a bi-gram representing a sequence of two words and values represent how many times the bi-gram was present for our documents.

8.4.3 TF-IDF Model

There are some potential problems which might arise with the Bag of Words model when it is used on large corpora. Since the feature vectors are based on absolute term frequencies, there might be some terms which occur frequently across all documents and these may tend to overshadow other terms in the feature set. The TF-IDF model tries to combat this issue by using a scaling or normalizing factor in its computation. TF-IDF stands for Term Frequency-Inverse Document Frequency, which uses a combination of two metrics in its computation, namely: *term frequency (tf)* and *inverse document frequency (idf)*. This technique was developed for ranking results for queries in search engines and now it is an indispensable model in the world of information retrieval and NLP.

Mathematically, we can define TF-IDF as tfidf = tf x idf, which can be expanded further to be represented as follows.

$$tfidf(w,D) = tf(w,D) \times idf(w,D) = tf(w,D) \times log\left(\frac{C}{df(w)}\right)$$

Here, ***tfidf(w, D)*** is the TF-IDF score for word ***w*** in document ***D***. The term ***tf(w, D)*** represents the term frequency of the word ***w*** in document ***D***, which can be obtained from the Bag of Words model. The term ***idf(w, D)*** is the inverse document frequency for the term ***w***, which can be computed as the log transform of the total number of documents in the corpus ***C*** divided by the document frequency of the word ***w***, which is basically the frequency of documents in the corpus where the word ***w*** occurs. There are multiple variants of this model but they all end up giving quite similar results. Let's apply this on our corpus now!

```
from sklearn.feature_extraction.text import TfidfVectorizer
tv = TfidfVectorizer(min_df=0., max_df=1., use_idf=True)
```

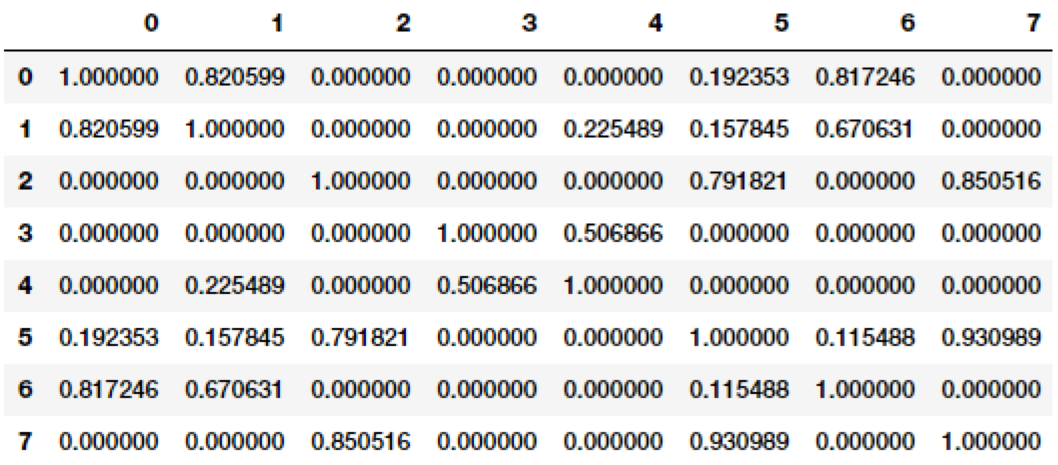
```
tv_matrix = tv.fit_transform(norm_corpus)

tv_matrix = tv_matrix.toarray()

vocab = tv.get_feature_names()

pd.DataFrame(np.round(tv_matrix, 2), columns=vocab)
```

8.5 Document Similarity

Document similarity is the process of using a distance or similarity based metric that can be used to identify how similar a text document is with any other document(s) based on features. Thus you can see that we can build on top of the tf-idf based features we engineered in the previous section and use them to generate new features which can be useful in domains like search engines, document clustering and information retrieval by leveraging these similarity based features.

Pairwise document similarity in a corpus involves computing document similarity for each pair of documents in a corpus. Thus if you have *C* documents in a corpus, you would end up with a *C* **x** *C* matrix such that each row and column represents the similarity score for a pair of documents, which represent the indices at the row and column, respectively. There are several similarity and distance metrics that are used to compute document similarity. These include cosine distance/similarity, euclidean distance, manhattan distance, BM25 similarity, jaccard distance and so on. In our analysis, we will be using perhaps the most popular and widely used similarity metric, cosine similarity and compare pairwise document similarity based on their TF-IDF feature vectors.

```
from sklearn.metrics.pairwise import cosine_similarity

similarity_matrix = cosine_similarity(tv_matrix)

similarity_df = pd.DataFrame(similarity_matrix)

similarity_df
```



	0	1	2	3	4	5	6	7
0	1.000000	0.820599	0.000000	0.000000	0.000000	0.192353	0.817246	0.000000
1	0.820599	1.000000	0.000000	0.000000	0.225489	0.157845	0.670631	0.000000
2	0.000000	0.000000	1.000000	0.000000	0.000000	0.791821	0.000000	0.850516
3	0.000000	0.000000	0.000000	1.000000	0.506866	0.000000	0.000000	0.000000
4	0.000000	0.225489	0.000000	0.506866	1.000000	0.000000	0.000000	0.000000
5	0.192353	0.157845	0.791821	0.000000	0.000000	1.000000	0.115488	0.930989
6	0.817246	0.670631	0.000000	0.000000	0.000000	0.115488	1.000000	0.000000
7	0.000000	0.000000	0.850516	0.000000	0.000000	0.930989	0.000000	1.000000

Cosine similarity basically gives us a metric representing the cosine of the angle between the feature vector representations of two text documents. Lower the angle between the documents, the closer and more similar they are as depicted in the following figure.

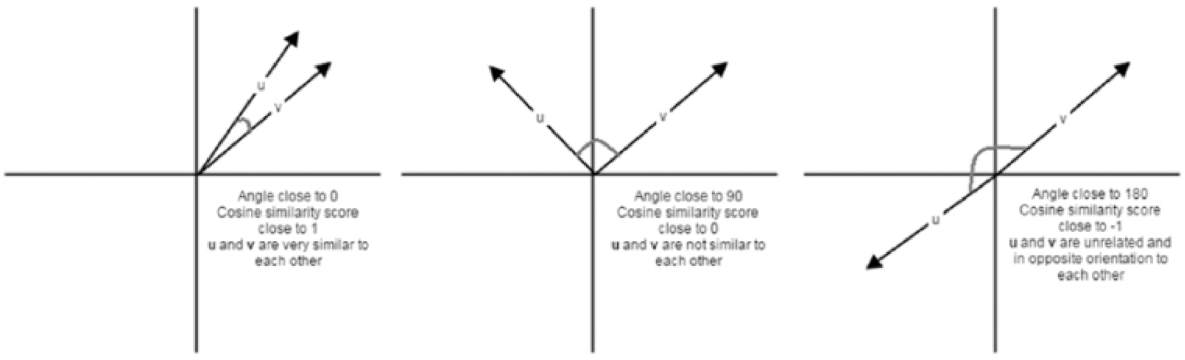
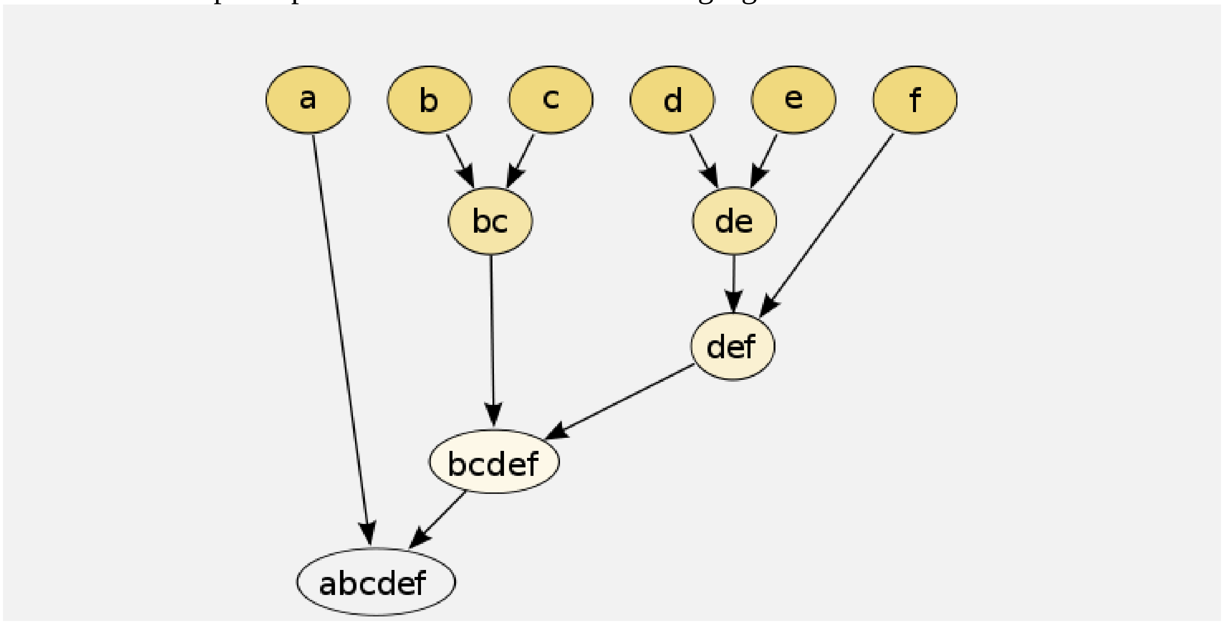

u
v
Angle close to 0
Cosine similarity score close to 1
u and v are very similar to each other
u
v
Angle close to 90
Cosine similarity score close to 0
u and v are not similar to each other
v
u
Angle close to 180
Cosine similarity score close to -1
u and v are unrelated and in opposite orientation to each other

Figure: Cosine similarity depictions for text document feature vectors

Looking closely at the similarity matrix clearly tells us that documents (0, 1 and 6), (2, 5 and 7) are very similar to one another and documents 3 and 4 are slightly similar to each other but the magnitude is not very strong, however still stronger than the other documents. This must indicate these similar documents have some similar features. This is a perfect example of grouping or clustering that can be solved by unsupervised learning especially when you are dealing with huge corpora of millions of text documents.

8.5.1 Document Clustering with Similarity Features

Clustering leverages unsupervised learning to group data points (documents in this scenario) into groups or clusters. We will be leveraging an unsupervised hierarchical clustering algorithm here to try and group similar documents from our toy corpus together by leveraging the document similarity features we generated earlier. There are two types of hierarchical clustering algorithms namely, agglomerative and divisive methods. We will be using a agglomerative clustering algorithm, which is hierarchical clustering using a bottom up approach i.e. each observation or document starts in its own cluster and clusters are successively merged together using a distance metric which measures distances between data points and a linkage merge criterion. A sample depiction is shown in the following figure.



Agglomerative Hierarchical Clustering

The selection of the linkage criterion governs the merge strategy. Some examples of linkage criteria are Ward, Complete linkage, Average linkage and so on. This criterion is very useful for choosing the pair of clusters (individual documents at the lowest step and clusters in higher steps) to merge at each step is based on the optimal value of an objective function. We choose the Ward's *minimum variance method* as our linkage criterion to minimize total within-cluster variance. Hence, at each step, we find the pair of clusters that leads to minimum increase in total within-cluster variance after merging. Since we already have our similarity features, let's build out the linkage matrix on our sample documents.