# MET CS 688 Feature Engineering II

Leila Ghaedi – Fall 2024

Creator: cemagraphics | Credit: Getty Images/iStockphoto

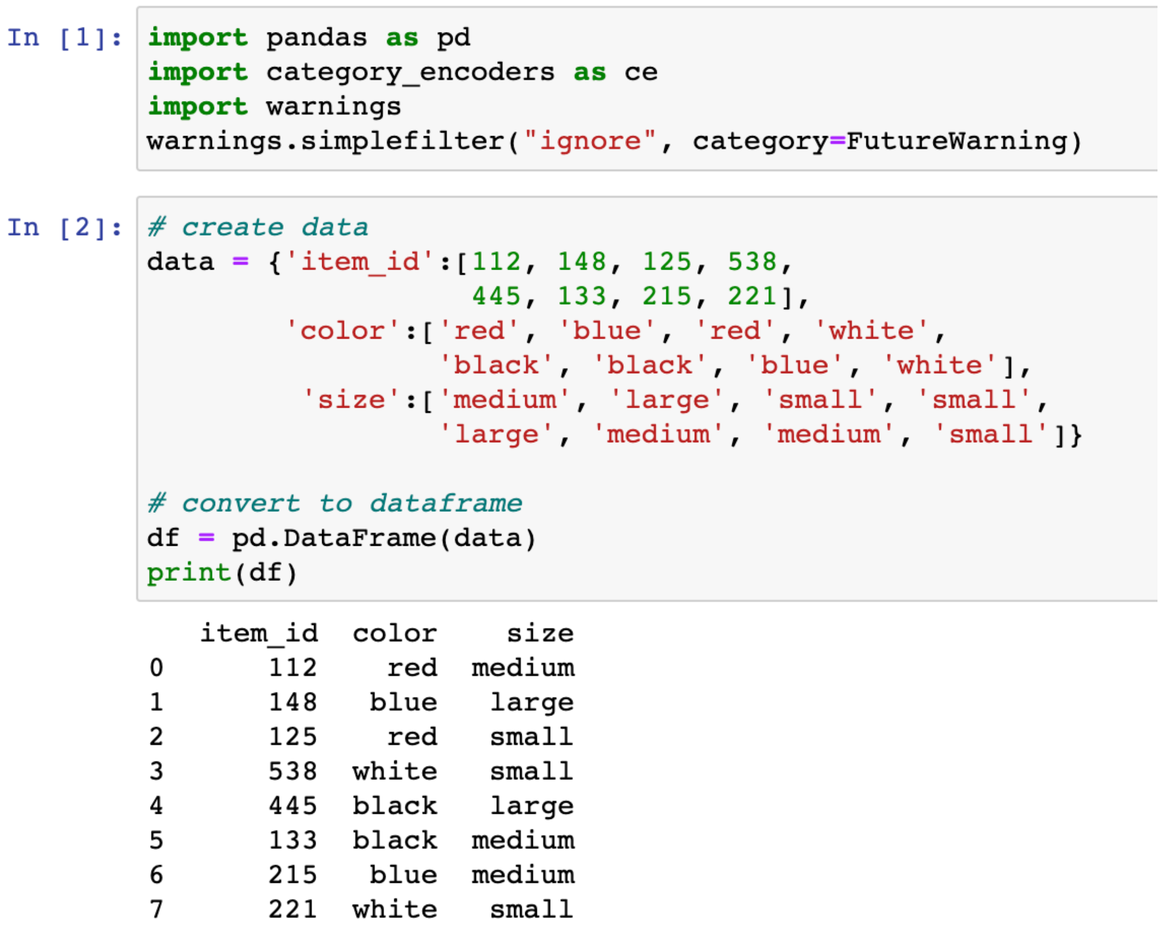
# Feature Engineering for Categorical Data

- Categorical data is non-numerical data that represents qualitative information, such as gender, color, or product categories. Before using categorical data in computational models, it must be preprocessed into numerical values. This process is called encoding.
- One_Hot Encoding: Converts any possible values of the data set into a bit(0,1).

# Feature Engineering for Categorical Data

- One_Hot Encoding:
  - Converts any possible values of the data set into a bit(0,1).
  - Represents a categorical value with k levels in k bits
- Dummy Encoding:
  - Is like one hot encoding only represents a categorical value with k levels in k-1 bits.
- Effect Encoding:
  - Is like dummy coding, but the reference category (all 0 bits) are presented with -1. It makes interpretation of results for linear regression easier.

# One Hot Encoding



In [1]:

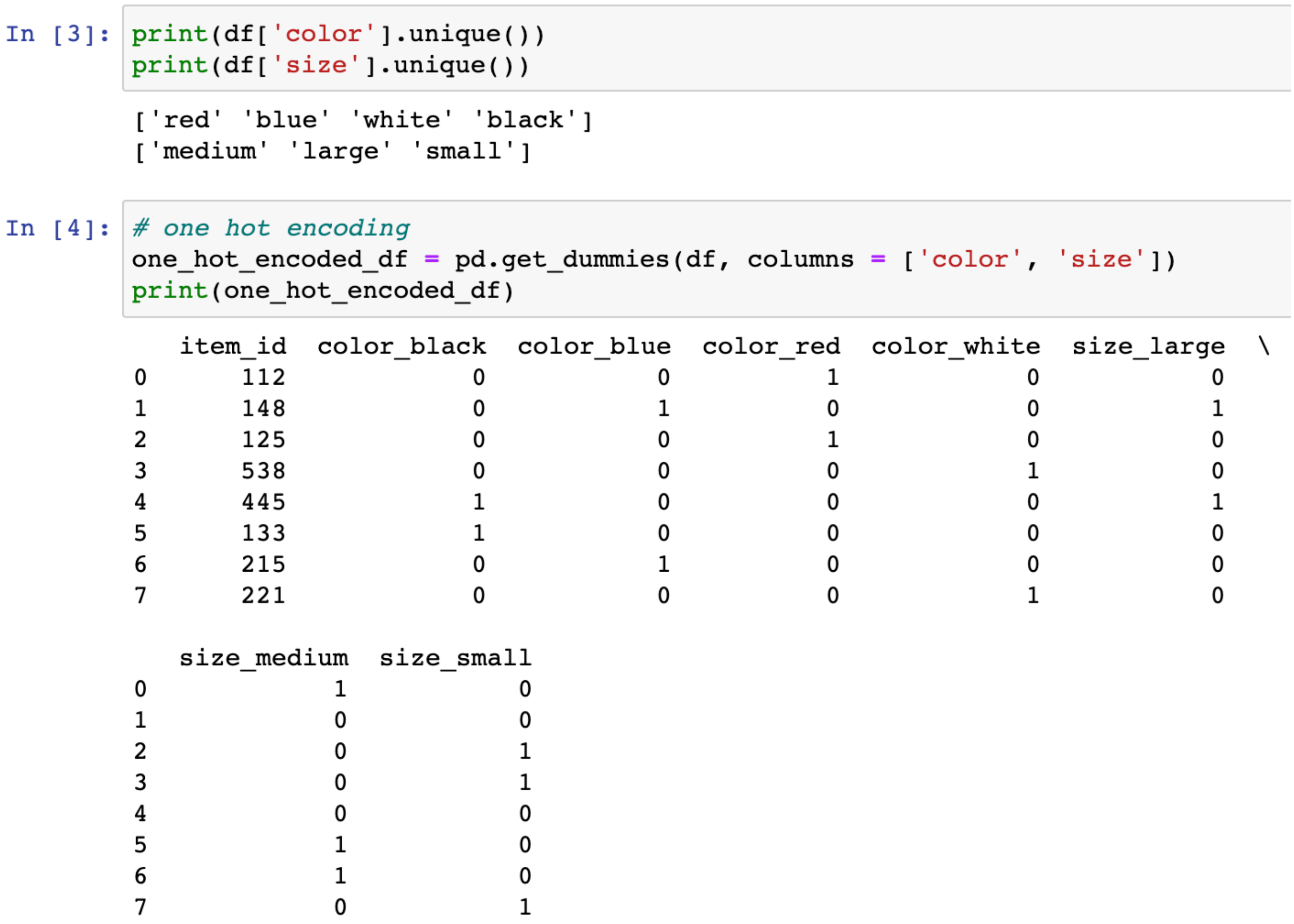
```
import pandas as pd
import category_encoders as ce
import warnings
warnings.simplefilter("ignore", category=FutureWarning)
```

In [2]:

```
# create data
data = {'item_id':[112, 148, 125, 538,
                   445, 133, 215, 221],
        'color':['red', 'blue', 'red', 'white',
                 'black', 'black', 'blue', 'white'],
         'size':['medium', 'large', 'small', 'small',
                 'large', 'medium', 'medium', 'small']}

# convert to dataframe
df = pd.DataFrame(data)
print(df)
```

```
   item_id  color    size
0      112    red  medium
1      148   blue   large
2      125    red   small
3      538  white   small
4      445  black   large
5      133  black  medium
6      215   blue  medium
7      221  white   small
```

# One Hot Encoding



In [3]:

```
print(df['color'].unique())
print(df['size'].unique())
```

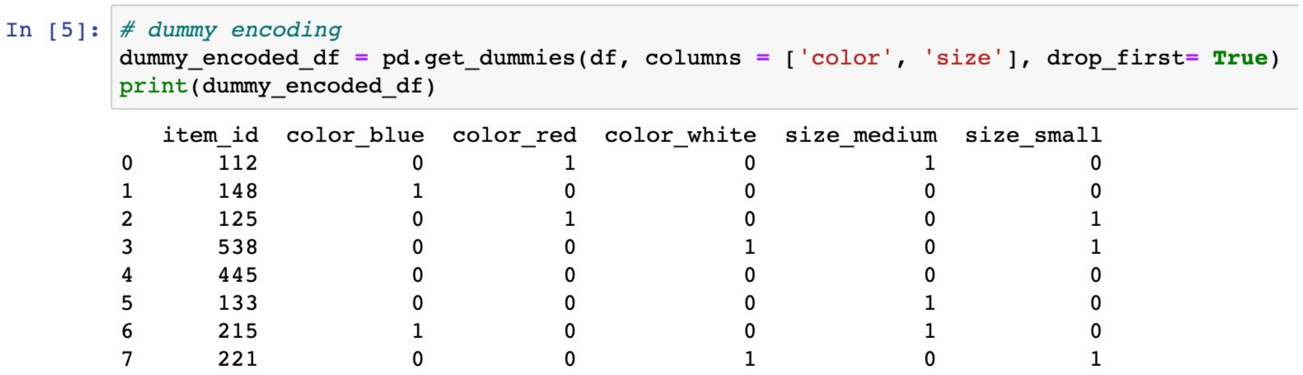
```
['red' 'blue' 'white' 'black']
['medium' 'large' 'small']
```

In [4]:

```
# one hot encoding
one_hot_encoded_df = pd.get_dummies(df, columns = ['color', 'size'])
print(one_hot_encoded_df)
```

```
   item_id  color_black  color_blue  color_red  color_white  size_large  \
0      112            0           0          1            0           0   
1      148            0           1          0            0           1   
2      125            0           0          1            0           0   
3      538            0           0          0            1           0   
4      445            1           0          0            0           1   
5      133            1           0          0            0           0   
6      215            0           1          0            0           0   
7      221            0           0          0            1           0   

   size_medium  size_small  
0            1           0  
1            0           0  
2            0           1  
3            0           1  
4            0           0  
5            1           0  
6            1           0  
7            0           1  
```

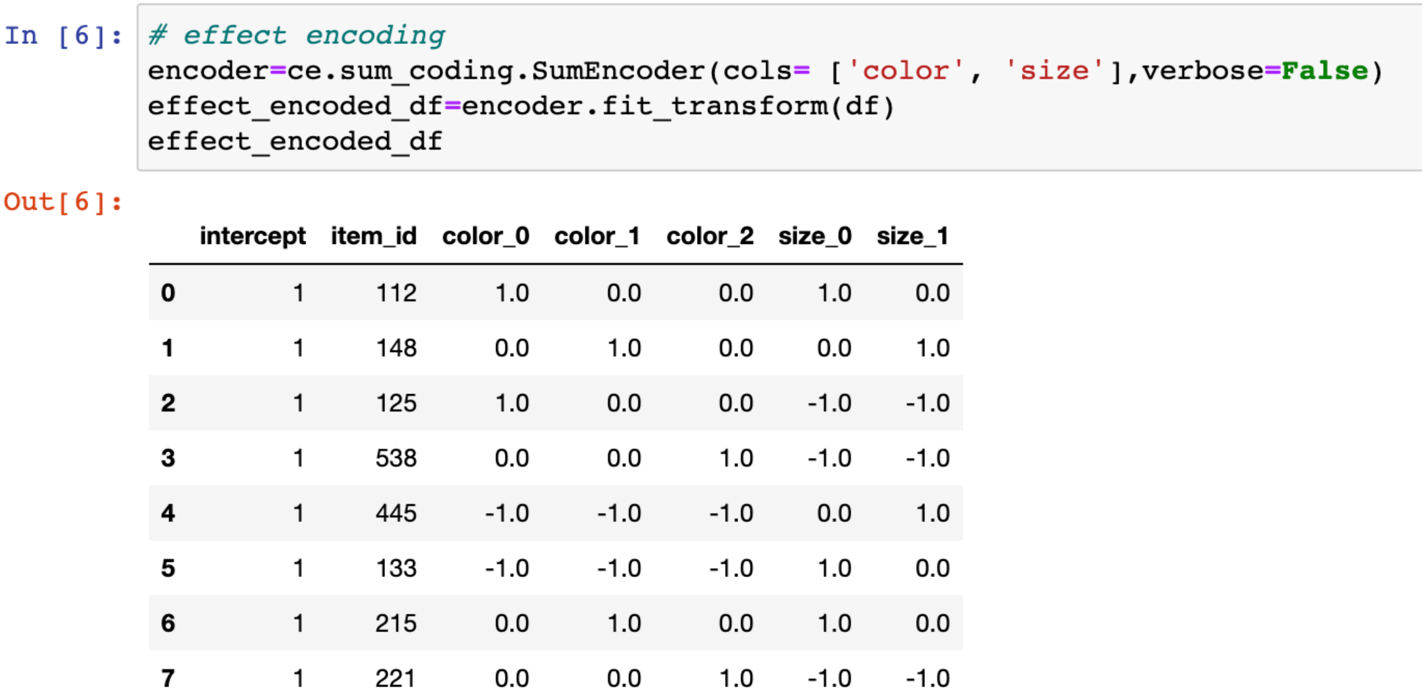
# Dummy Encoding



```
In [5]: # dummy encoding
        dummy_encoded_df = pd.get_dummies(df, columns = ['color', 'size'], drop_first= True)
        print(dummy_encoded_df)
```

```
   item_id  color_blue  color_red  color_white  size_medium  size_small
0      112           0          1            0            1           0
1      148           1          0            0            0           0
2      125           0          1            0            0           1
3      538           0          0            1            0           1
4      445           0          0            0            0           0
5      133           0          0            0            1           0
6      215           1          0            0            1           0
7      221           0          0            1            0           1
```

# Effect Encoding



In [6]:

```
# effect encoding
encoder=ce.sum_coding.SumEncoder(cols= ['color', 'size'],verbose=False)
effect_encoded_df=encoder.fit_transform(df)
effect_encoded_df
```

Out[6]:

|  | intercept | item_id | color_0 | color_1 | color_2 | size_0 | size_1 |
|---|---|---|---|---|---|---|---|
| **0** | 1 | 112 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **1** | 1 | 148 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| **2** | 1 | 125 | 1.0 | 0.0 | 0.0 | -1.0 | -1.0 |
| **3** | 1 | 538 | 0.0 | 0.0 | 1.0 | -1.0 | -1.0 |
| **4** | 1 | 445 | -1.0 | -1.0 | -1.0 | 0.0 | 1.0 |
| **5** | 1 | 133 | -1.0 | -1.0 | -1.0 | 1.0 | 0.0 |
| **6** | 1 | 215 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 |
| **7** | 1 | 221 | 0.0 | 0.0 | 1.0 | -1.0 | -1.0 |

# Extracting Features from Text Variables

# Text (Unstructured Data)

- Text data is unstructured, which means it does not follow a pattern, like the tabular pattern.
- Text may also vary in length and content, and the writing style may be different. How can we extract information from text variables to inform our predictive models?
- The techniques we will cover here belong to the realm of Natural Language Processing (NLP). NLP is a subfield of linguistics and computer science, concerned with the interactions between computer and human language.
- NLP includes a multitude of techniques to understand the syntax, semantics, and discourse of text.

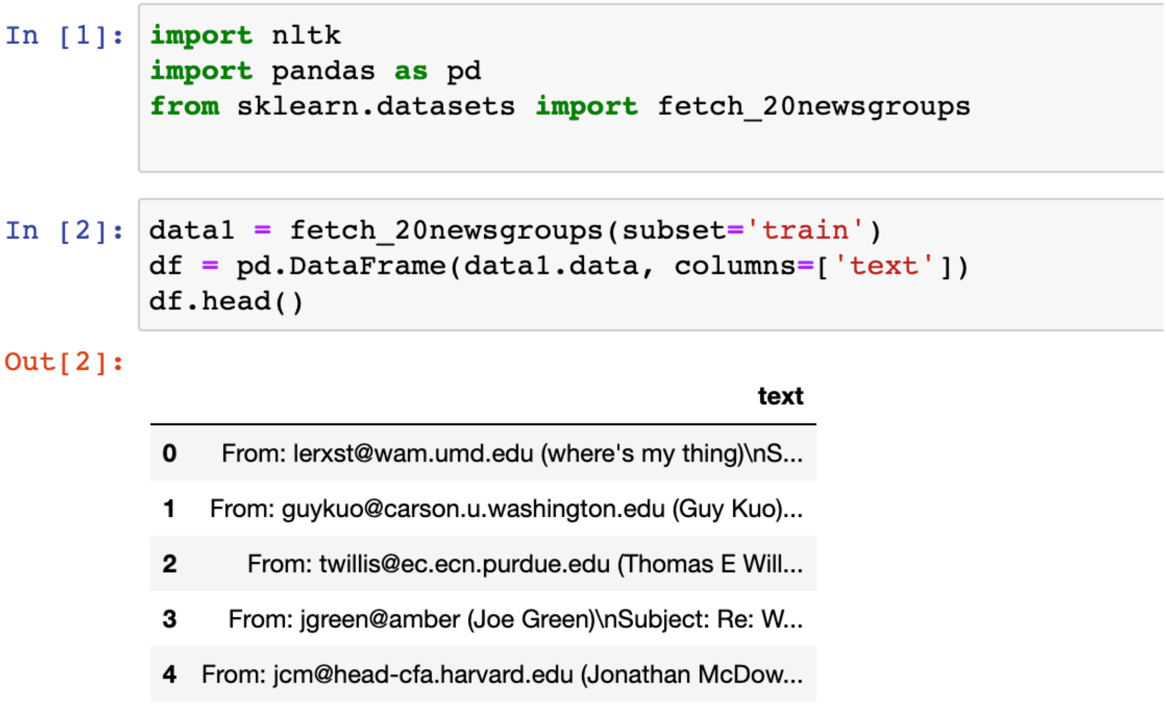
# Text Feature Engineering

- Here we discuss techniques that allow us to quickly extract features from short pieces of text, in the context of providing input to predictive models.
- Specifically, we will discuss how to capture text complexity by looking at some statistical parameters of the text such as the word length and count, the number of words and unique words used, the number of sentences, and so on.
- Our tools:
  - pandas
  - scikit-learn libraries
  - Natural Language Toolkit (NLTK)

# Text Feature Engineering

- We can capture text complexity by extracting the following information:
  - The total number of characters in the text
  - The total number of words
  - The total number of unique words
  - Lexical diversity = total number of words / number of unique words Word average length = number of characters / number of words

# Text Feature Engineering Example



In [1]:

```python
import nltk
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
```

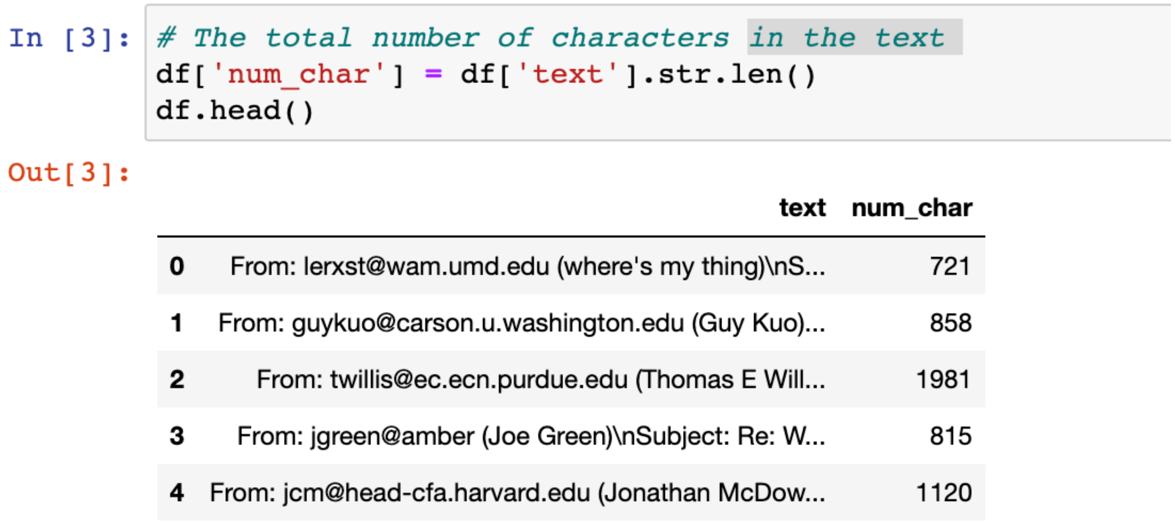
In [2]:

```python
data1 = fetch_20newsgroups(subset='train')
df = pd.DataFrame(data1.data, columns=['text'])
df.head()
```

Out[2]:

| | text |
|---|---|
| 0 | From: lerxst@wam.umd.edu (where's my thing)\nS... |
| 1 | From: guykuo@carson.u.washington.edu (Guy Kuo)... |
| 2 | From: twillis@ec.ecn.purdue.edu (Thomas E Will... |
| 3 | From: jgreen@amber (Joe Green)\nSubject: Re: W... |
| 4 | From: jcm@head-cfa.harvard.edu (Jonathan McDow... |

# Text Feature Engineering Example



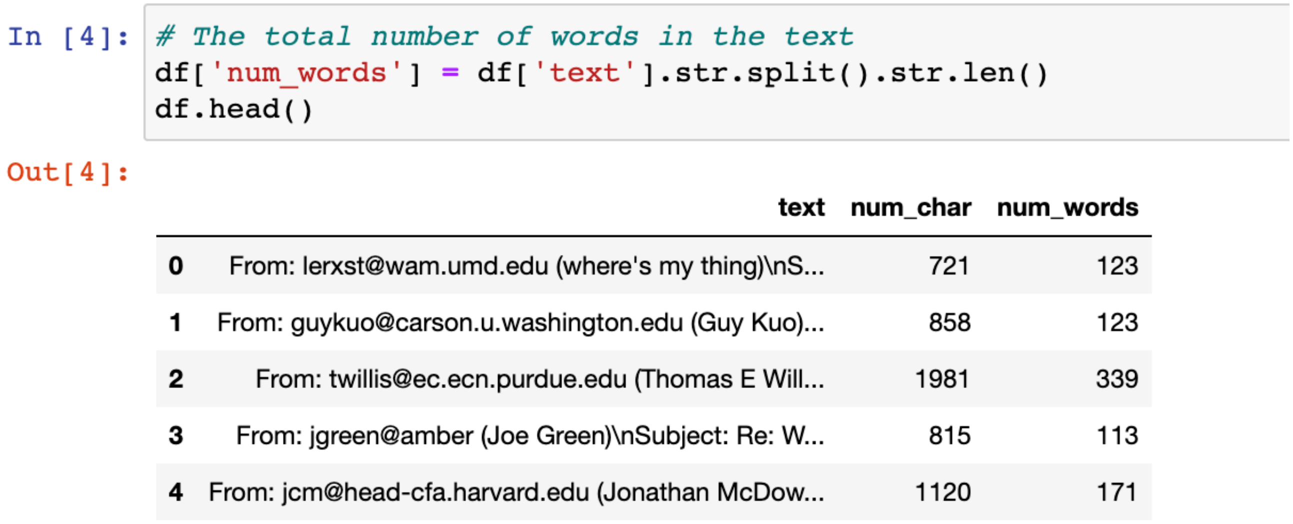
In [3]:

```
# The total number of characters in the text
df['num_char'] = df['text'].str.len()
df.head()
```

Out[3]:

| | text | num_char |
|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 |

# Text Feature Engineering Example



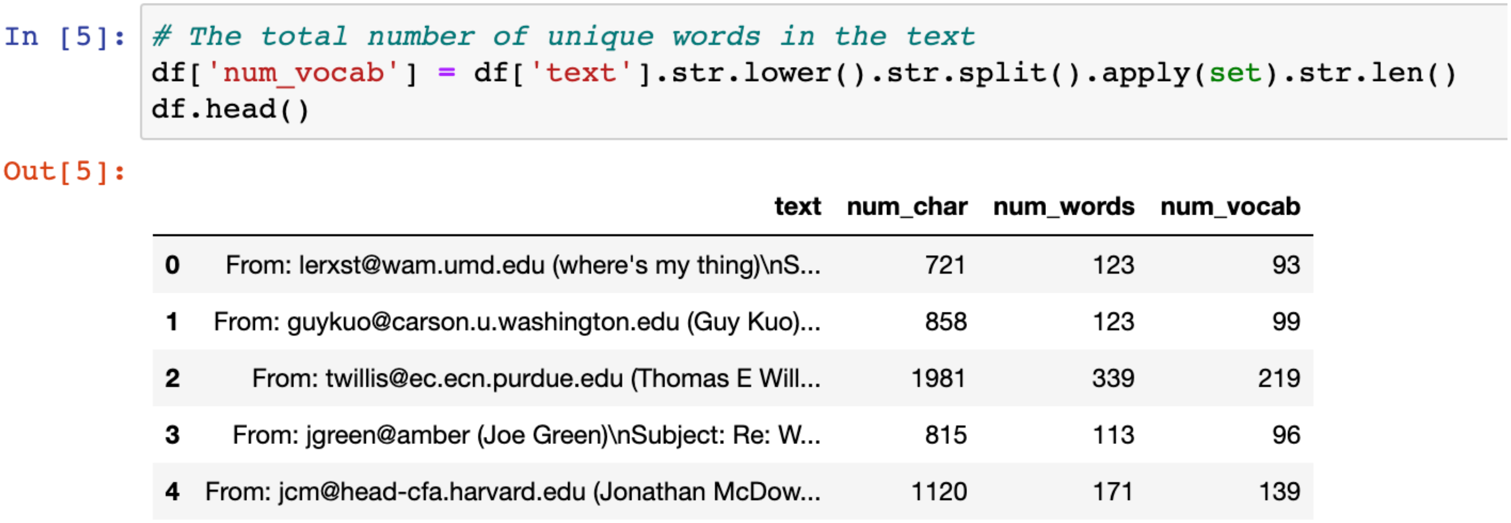
In [4]:

```
# The total number of words in the text
df['num_words'] = df['text'].str.split().str.len()
df.head()
```

Out[4]:

| | text | num_char | num_words |
|---|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 | 123 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 | 123 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 | 339 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 | 113 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 | 171 |

# Text Feature Engineering Example



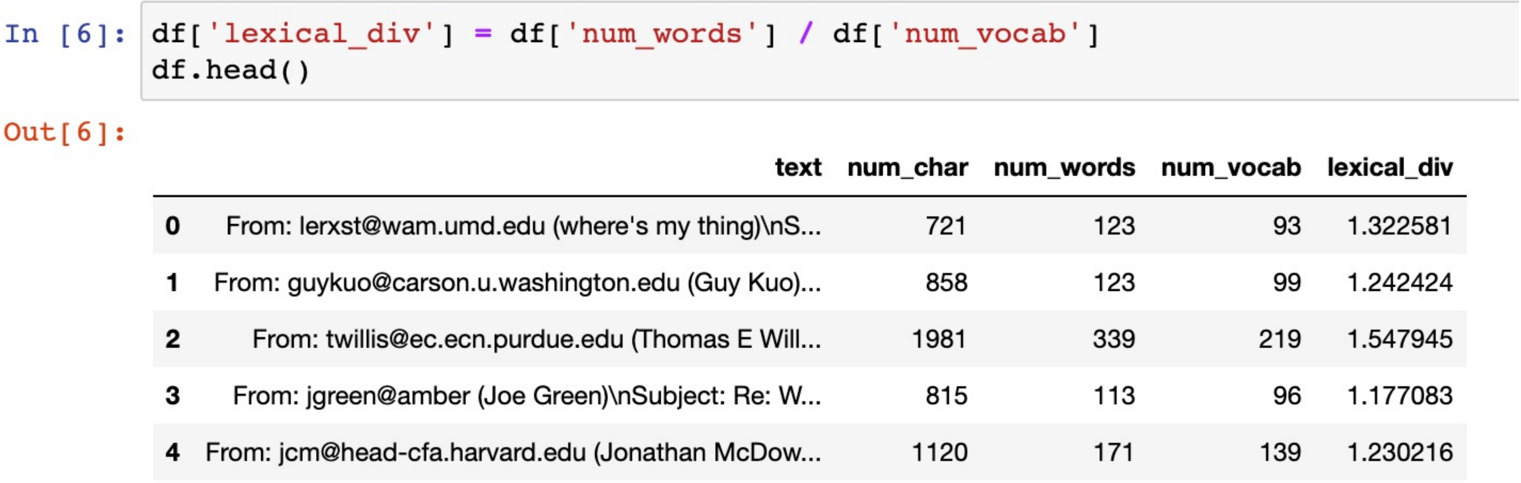
In [5]:

```
# The total number of unique words in the text
df['num_vocab'] = df['text'].str.lower().str.split().apply(set).str.len()
df.head()
```

Out[5]:

| | text | num_char | num_words | num_vocab |
|---|---|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 | 123 | 93 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 | 123 | 99 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 | 339 | 219 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 | 113 | 96 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 | 171 | 139 |

# Text Feature Engineering Example



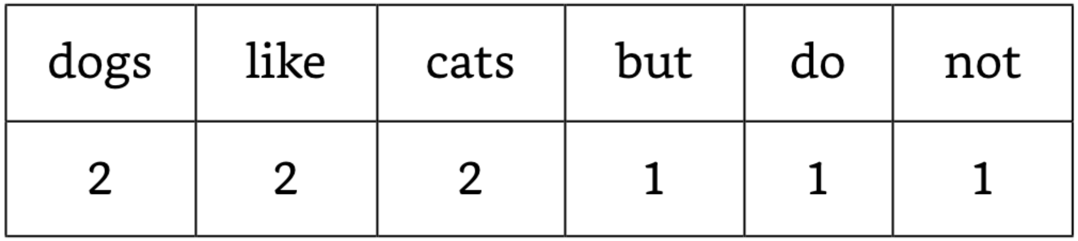
```
In [6]: df['lexical_div'] = df['num_words'] / df['num_vocab']
        df.head()
```

Out[6]:

| | text | num_char | num_words | num_vocab | lexical_div |
|---|---|---|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 | 123 | 93 | 1.322581 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 | 123 | 99 | 1.242424 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 | 339 | 219 | 1.547945 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 | 113 | 96 | 1.177083 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 | 171 | 139 | 1.230216 |

# Bag of Words (BoW)

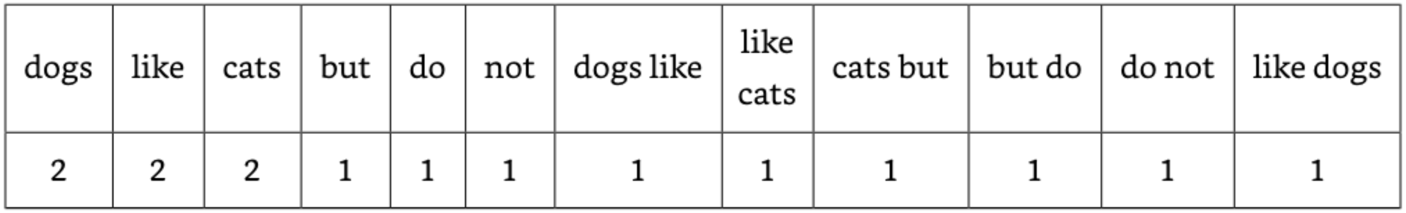
- A bag-of-words (BoW), is a simplified representation of a text that captures the words that are present in the text and the number of times each word appears in the text.
- Example: “Dogs like cats, but cats do not like dogs”



| dogs | like | cats | but | do | not |
|---|---|---|---|---|---|
| 2 | 2 | 2 | 1 | 1 | 1 |

# n-grams

- To capture some syntax, BoW can be used together with n-grams. An n-gram is a contiguous sequence of n items in a given text. Continuing with the sentence “Dogs like cats, but cats do not like dogs”, the derived 2-grams are as follows:
  - Dogs like
  - like cats
  - cats but
  - but do
  - do not
  - like dogs

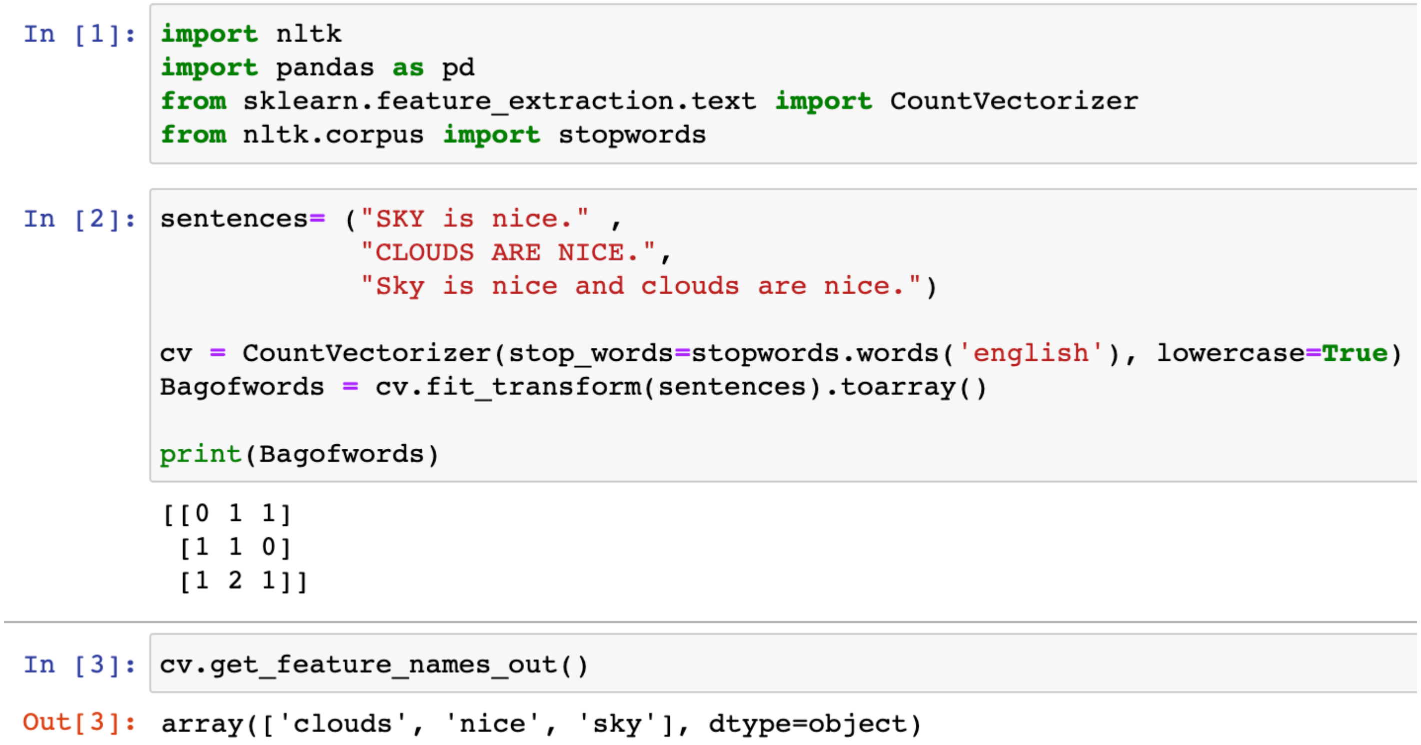


| dogs | like | cats | but | do | not | dogs like | like cats | cats but | but do | do not | like dogs |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# Bag of Words (BoW), n_grams

- CountVectorizer() is a transformer from scikit-learn which sets words in lowercase, removes stop words, retains words with a minimum acceptable frequency, and capture n-grams all together.

# Example BoW



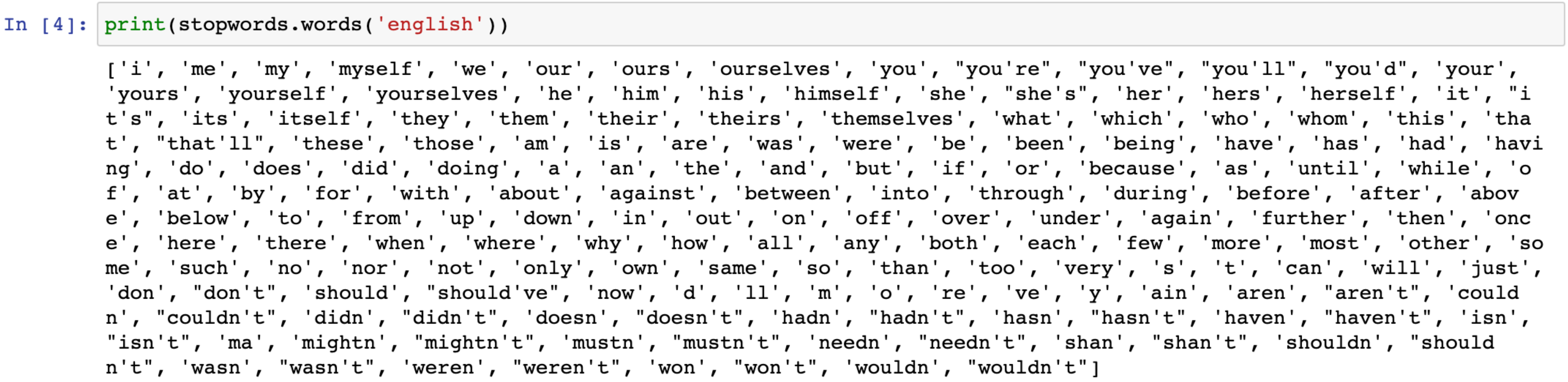
```
In [1]: import nltk
        import pandas as pd
        from sklearn.feature_extraction.text import CountVectorizer
        from nltk.corpus import stopwords
```

```
In [2]: sentences= ("SKY is nice." ,
                    "CLOUDS ARE NICE.",
                    "Sky is nice and clouds are nice.")

        cv = CountVectorizer(stop_words=stopwords.words('english'), lowercase=True)
        Bagofwords = cv.fit_transform(sentences).toarray()

        print(Bagofwords)

        [[0 1 1]
         [1 1 0]
         [1 2 1]]
```

```
In [3]: cv.get_feature_names_out()

Out[3]: array(['clouds', 'nice', 'sky'], dtype=object)
```

# Stop Words

Stop words are very frequently used words to make sentences flow, but that, per se, do not carry any useful information.



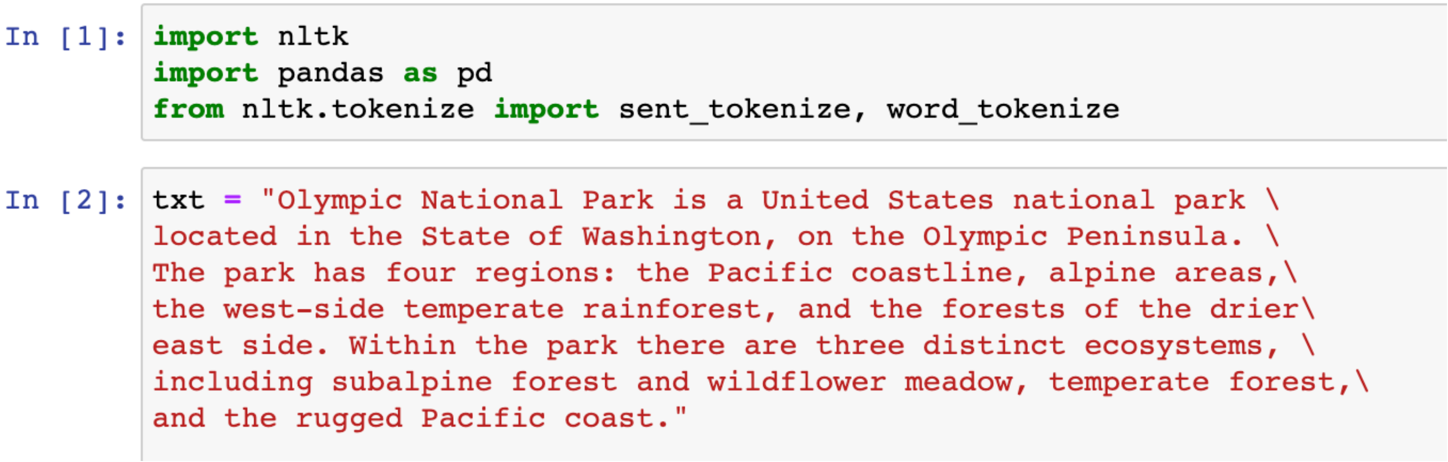
```
In [4]: print(stopwords.words('english'))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your',
'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "i
t's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'tha
t', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'havi
ng', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'o
f', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'abov
e', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'onc
e', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'so
me', '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', 'aren', "aren't", 'could
n', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven'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"]
```

# Tokenization

- Tokenization is used in NLP to split paragraphs and sentences into smaller units that can be more easily processed.
- The first step of the NLP process is gathering the data (text) and breaking it into understandable parts (sentences, words).

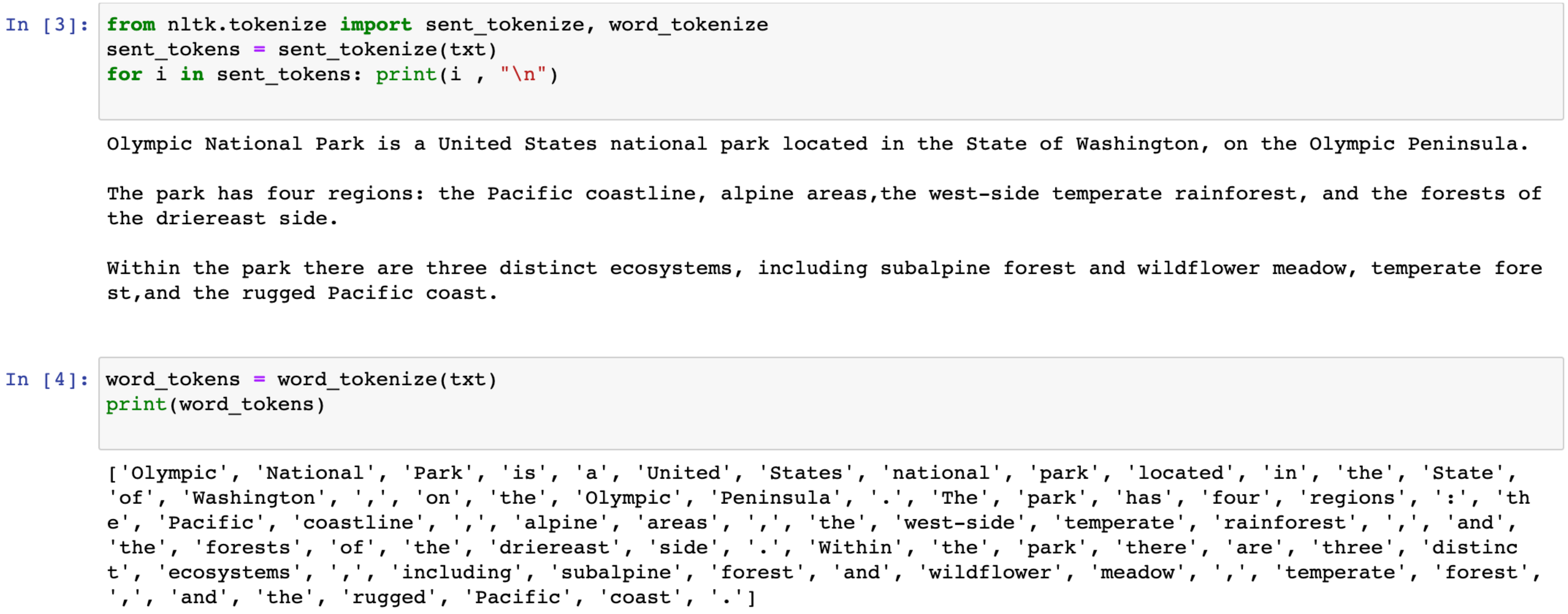
# Tokenization Example



```
In [1]: import nltk
        import pandas as pd
        from nltk.tokenize import sent_tokenize, word_tokenize
```

```
In [2]: txt = "Olympic National Park is a United States national park \
        located in the State of Washington, on the Olympic Peninsula. \
        The park has four regions: the Pacific coastline, alpine areas,\
        the west-side temperate rainforest, and the forests of the drier\
        east side. Within the park there are three distinct ecosystems, \
        including subalpine forest and wildflower meadow, temperate forest,\
        and the rugged Pacific coast."
```

# Tokenization Example



```
In [3]: from nltk.tokenize import sent_tokenize, word_tokenize
        sent_tokens = sent_tokenize(txt)
        for i in sent_tokens: print(i , "\n")
```

```
Olympic National Park is a United States national park located in the State of Washington, on the Olympic Peninsula.

The park has four regions: the Pacific coastline, alpine areas,the west-side temperate rainforest, and the forests of
the driereast side.

Within the park there are three distinct ecosystems, including subalpine forest and wildflower meadow, temperate fore
st,and the rugged Pacific coast.
```

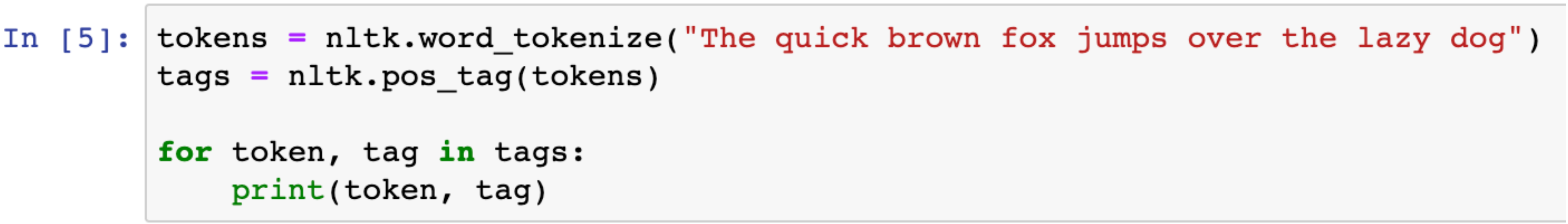
```
In [4]: word_tokens = word_tokenize(txt)
        print(word_tokens)
```

```
['Olympic', 'National', 'Park', 'is', 'a', 'United', 'States', 'national', 'park', 'located', 'in', 'the', 'State',
'of', 'Washington', ',', 'on', 'the', 'Olympic', 'Peninsula', '.', 'The', 'park', 'has', 'four', 'regions', ':', 'th
e', 'Pacific', 'coastline', ',', 'alpine', 'areas', ',', 'the', 'west-side', 'temperate', 'rainforest', ',', 'and',
'the', 'forests', 'of', 'the', 'driereast', 'side', '.', 'Within', 'the', 'park', 'there', 'are', 'three', 'distinc
t', 'ecosystems', ',', 'including', 'subalpine', 'forest', 'and', 'wildflower', 'meadow', ',', 'temperate', 'forest',
',', 'and', 'the', 'rugged', 'Pacific', 'coast', '.']
```

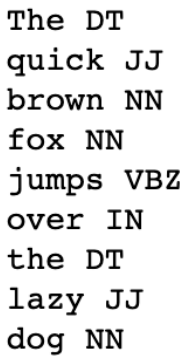
# (Part of Speech) PoS Tagging

- Part of speech tagging is an NLP technique that assigns a part of speech tag to each word in a sentence. The part of speech tag indicates the grammatical category of the word, such as noun, verb, adjective, adverb, etc.

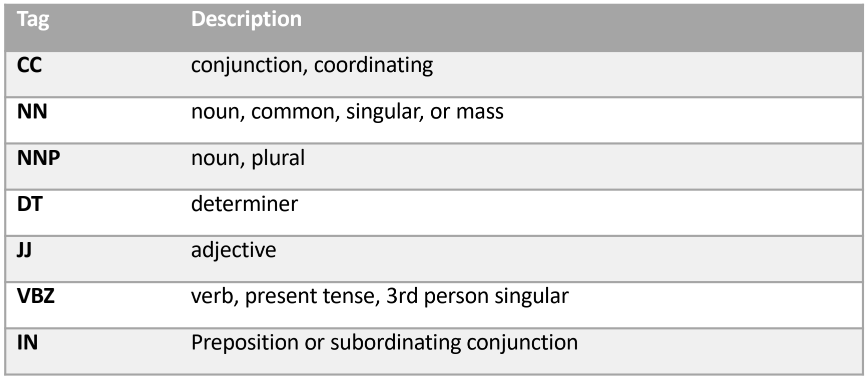
# PoS Tagging Example



```
In [5]: tokens = nltk.word_tokenize("The quick brown fox jumps over the lazy dog")
        tags = nltk.pos_tag(tokens)

        for token, tag in tags:
            print(token, tag)
```



```
The DT
quick JJ
brown NN
fox NN
jumps VBZ
over IN
the DT
lazy JJ
dog NN
```



| Tag | Description |
|---|---|
| CC | conjunction, coordinating |
| NN | noun, common, singular, or mass |
| NNP | noun, plural |
| DT | determiner |
| JJ | adjective |
| VBZ | verb, present tense, 3rd person singular |
| IN | Preposition or subordinating conjunction |

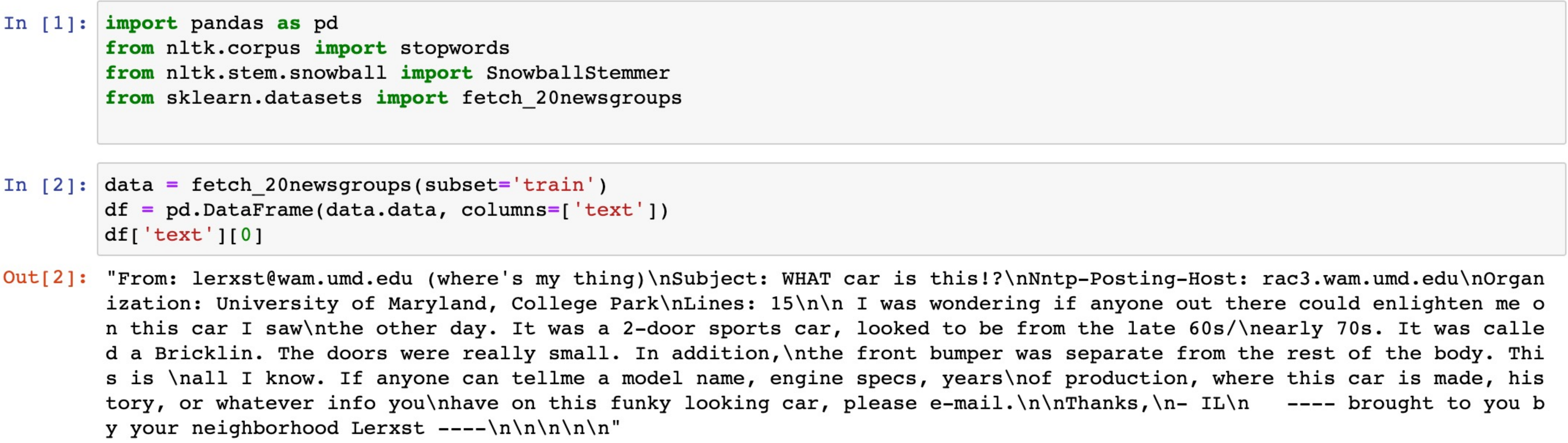
# Text Preprocessing Steps

- Text cleaning or text preprocessing involves:
  - Punctuation removal
  - Elimination of stop words
  - Character case setting
  - Word stemming

# Word Stemming

- Word stemming refers to reducing each word to its root or base so that the words playing, plays, and played become play, which, in essence, convey the same or very similar meaning.
    - had —> have
    - bananas —> banana
    - swimming —> swim

# Text Preprocessing Example

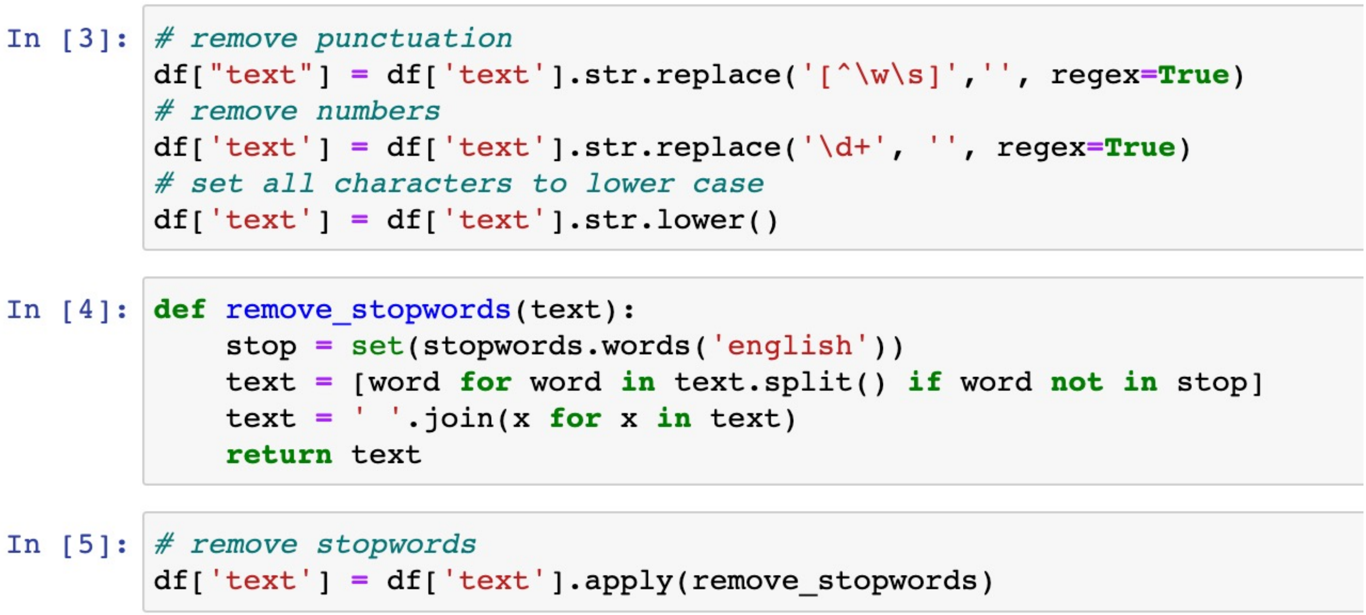


```
In [1]: import pandas as pd
        from nltk.corpus import stopwords
        from nltk.stem.snowball import SnowballStemmer
        from sklearn.datasets import fetch_20newsgroups
```

```
In [2]: data = fetch_20newsgroups(subset='train')
        df = pd.DataFrame(data.data, columns=['text'])
        df['text'][0]
```

```
Out[2]: "From: lerxst@wam.umd.edu (where's my thing)\nSubject: WHAT car is this!?\nNntp-Posting-Host: rac3.wam.umd.edu\nOrgan
        ization: University of Maryland, College Park\nLines: 15\n\n I was wondering if anyone out there could enlighten me o
        n this car I saw\nthe other day. It was a 2-door sports car, looked to be from the late 60s/\nearly 70s. It was calle
        d a Bricklin. The doors were really small. In addition,\nthe front bumper was separate from the rest of the body. Thi
        s is \nall I know. If anyone can tellme a model name, engine specs, years\nof production, where this car is made, his
        tory, or whatever info you\nhave on this funky looking car, please e-mail.\n\nThanks,\n- IL\n   ---- brought to you b
        y your neighborhood Lerxst ----\n\n\n\n\n"
```

# Text Preprocessing Example

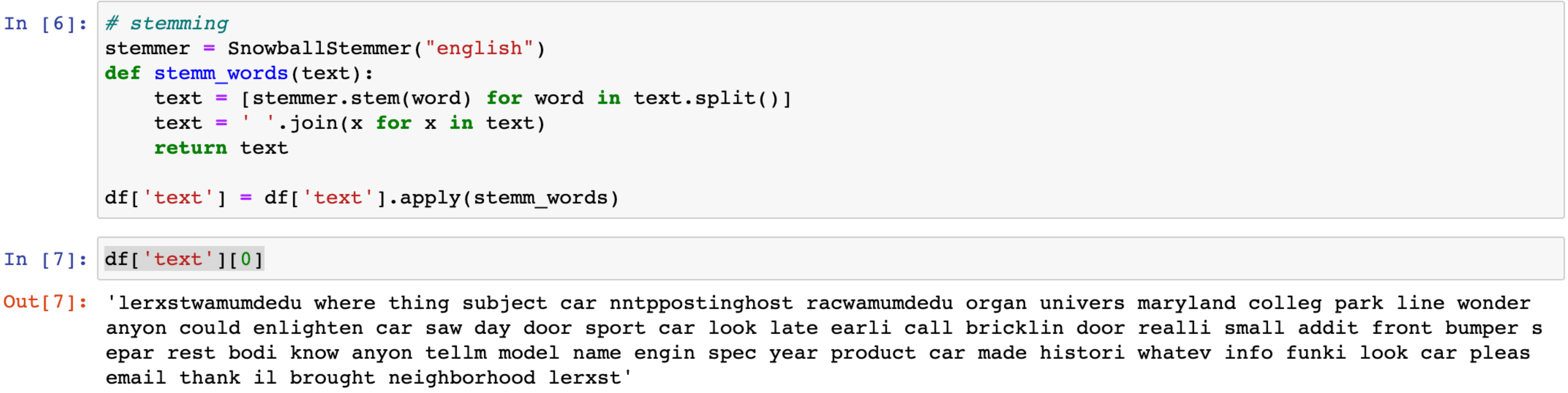


```
In [3]: # remove punctuation
        df["text"] = df['text'].str.replace('[^\w\s]','', regex=True)
        # remove numbers
        df['text'] = df['text'].str.replace('\d+', '', regex=True)
        # set all characters to lower case
        df['text'] = df['text'].str.lower()
```

```
In [4]: def remove_stopwords(text):
            stop = set(stopwords.words('english'))
            text = [word for word in text.split() if word not in stop]
            text = ' '.join(x for x in text)
            return text
```

```
In [5]: # remove stopwords
        df['text'] = df['text'].apply(remove_stopwords)
```

# Text Preprocessing Example



In [6]:
```
# stemming
stemmer = SnowballStemmer("english")
def stemm_words(text):
    text = [stemmer.stem(word) for word in text.split()]
    text = ' '.join(x for x in text)
    return text

df['text'] = df['text'].apply(stemm_words)
```

In [7]:
```
df['text'][0]
```

Out[7]:
```
'lerxstwamumdedu where thing subject car nntppostinghost racwamumdedu organ univers maryland colleg park line wonder
anyon could enlighten car saw day door sport car look late earli call bricklin door realli small addit front bumper s
epar rest bodi know anyon tellm model name engin spec year product car made histori whatev info funki look car pleas
email thank il brought neighborhood lerxst'
```

# Text Preprocessing

- The cleaning steps performed, resulted in strings containing the original text, without punctuation or numbers, in lowercase, without common words, and with the root of the word instead of its conjugated form.
- The data, as it is returned, can be used to derive features as described in the counting characters, words, and vocabulary or to create BoWs and TI-IDF matrices.

# Term Frequency-Inverse Document Frequency (TF-IDF)

- TF-IDF is a numerical statistic that captures how relevant a word is in a document, with respect to the entire collection of documents.
- Some words will appear a lot within a text document as well as across documents, for example, the English words the, a, and is. These words generally convey little information about the actual content of the document and don't make it stand out of the crowd.
- TF-IDF provides a way to weigh the importance of a word, by examining how many times it appears in a document, with respect to how often it appears across documents.
- Hence, commonly occurring words such as the, a, and is will have a low weight, and words more specific to a topic, such as leopard, will have a higher weight.

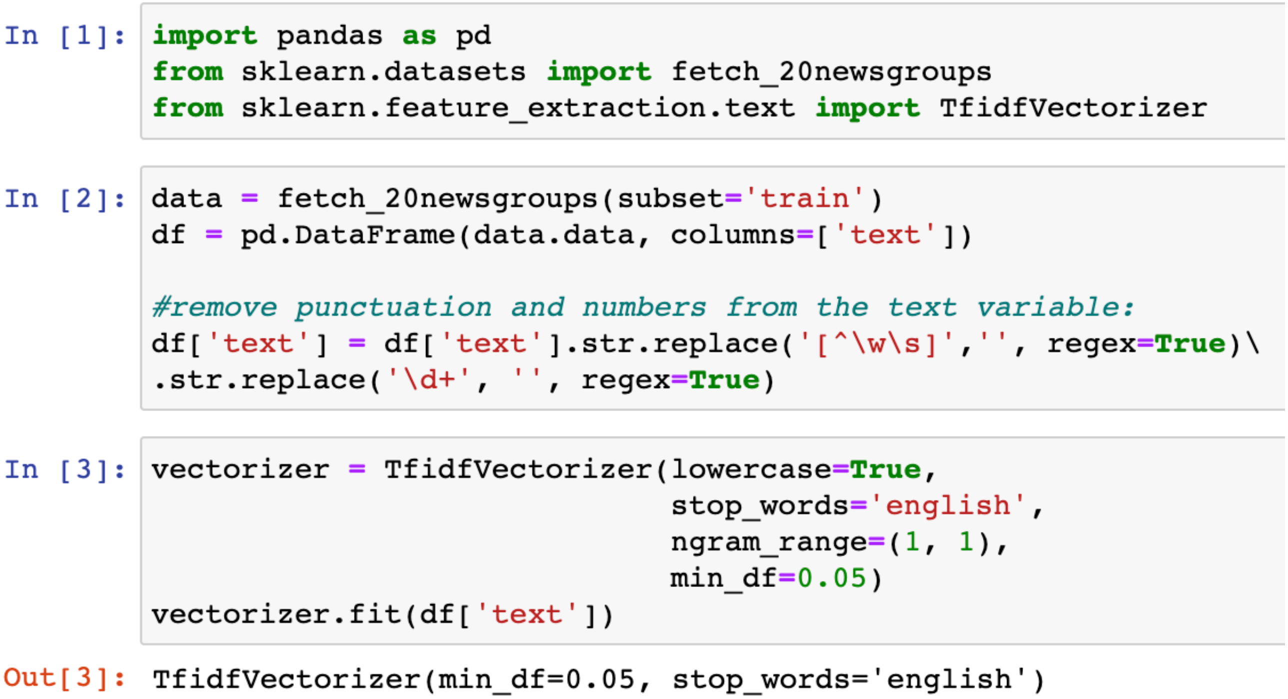
# TF-IDF

- TF-IDF = term frequency * inverse document frequency
- Term frequency is, the count of the word in an individual text. So, for term t, the term frequency is calculated as tf(t) = count(t) and is determined text by text.
- The inverse document frequency is a measure of how common the word is across all documents and is usually calculated on a logarithmic scale.

$$idf(t) = log(\frac{n}{1 + df(t)})$$

- Here, n is the total number of documents and df(t) the number of documents in which the term t appears.

# TF-IDF Example

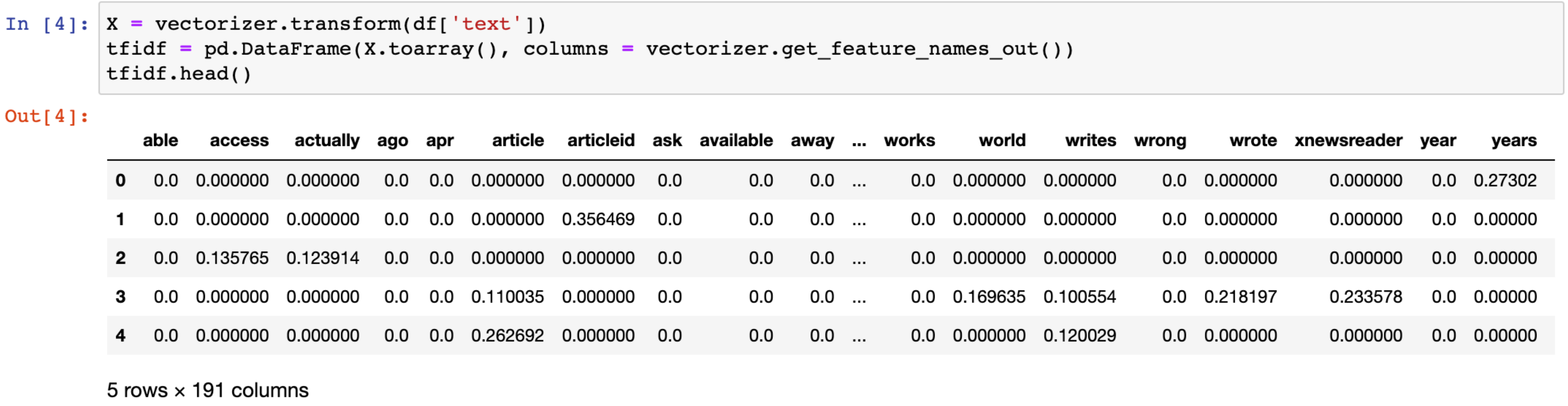


```
In [1]: import pandas as pd
        from sklearn.datasets import fetch_20newsgroups
        from sklearn.feature_extraction.text import TfidfVectorizer
```

```
In [2]: data = fetch_20newsgroups(subset='train')
        df = pd.DataFrame(data.data, columns=['text'])

        #remove punctuation and numbers from the text variable:
        df['text'] = df['text'].str.replace('[^\w\s]','', regex=True)\
        .str.replace('\d+', '', regex=True)
```

```
In [3]: vectorizer = TfidfVectorizer(lowercase=True,
                                     stop_words='english',
                                     ngram_range=(1, 1),
                                     min_df=0.05)
        vectorizer.fit(df['text'])

Out[3]: TfidfVectorizer(min_df=0.05, stop_words='english')
```

# TF-IDF Example



In [4]:

```
X = vectorizer.transform(df['text'])
tfidf = pd.DataFrame(X.toarray(), columns = vectorizer.get_feature_names_out())
tfidf.head()
```

Out[4]:

| | able | access | actually | ago | apr | article | articleid | ask | available | away | ... | works | world | writes | wrong | wrote | xnewsreader | year | years |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.27302 |
| **1** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.000000 | 0.356469 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.00000 |
| **2** | 0.0 | 0.135765 | 0.123914 | 0.0 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.00000 |
| **3** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.110035 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.169635 | 0.100554 | 0.0 | 0.218197 | 0.233578 | 0.0 | 0.00000 |
| **4** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.262692 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.120029 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.00000 |

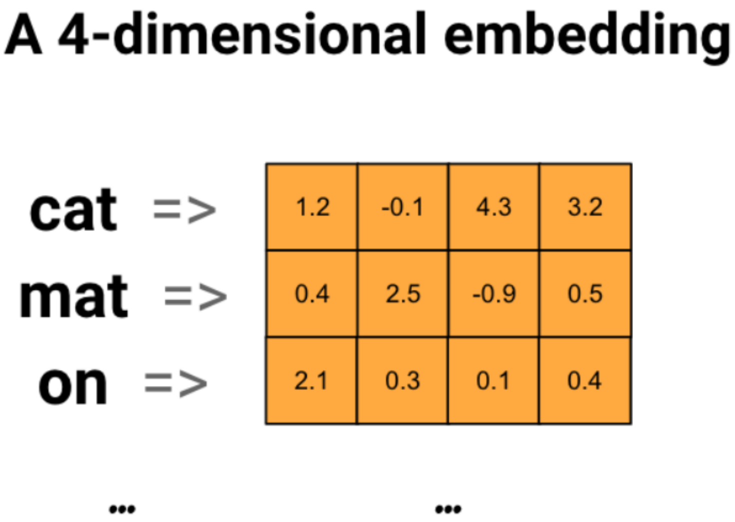
5 rows × 191 columns

# Word Embedding

- Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding.
- Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer).
- It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets.

# Word Embedding

- Each word is represented as a 4-dimensional vector of floating point values.
- Another way to think of an embedding is as "lookup table".
- After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.



A 4-dimensional embedding

| | | | | |
|---|---|---|---|---|
| cat => | 1.2 | -0.1 | 4.3 | 3.2 |
| mat => | 0.4 | 2.5 | -0.9 | 0.5 |
| on => | 2.1 | 0.3 | 0.1 | 0.4 |
| ... | ... | | | |

# Extracting Features from Text Variables

# Text (Unstructured Data)

- Text data is unstructured, which means it does not follow a pattern, like the tabular pattern.
- Text may also vary in length and content, and the writing style may be different. How can we extract information from text variables to inform our predictive models?
- The techniques we will cover here belong to the realm of Natural Language Processing (NLP). NLP is a subfield of linguistics and computer science, concerned with the interactions between computer and human language.
- NLP includes a multitude of techniques to understand the syntax, semantics, and discourse of text.

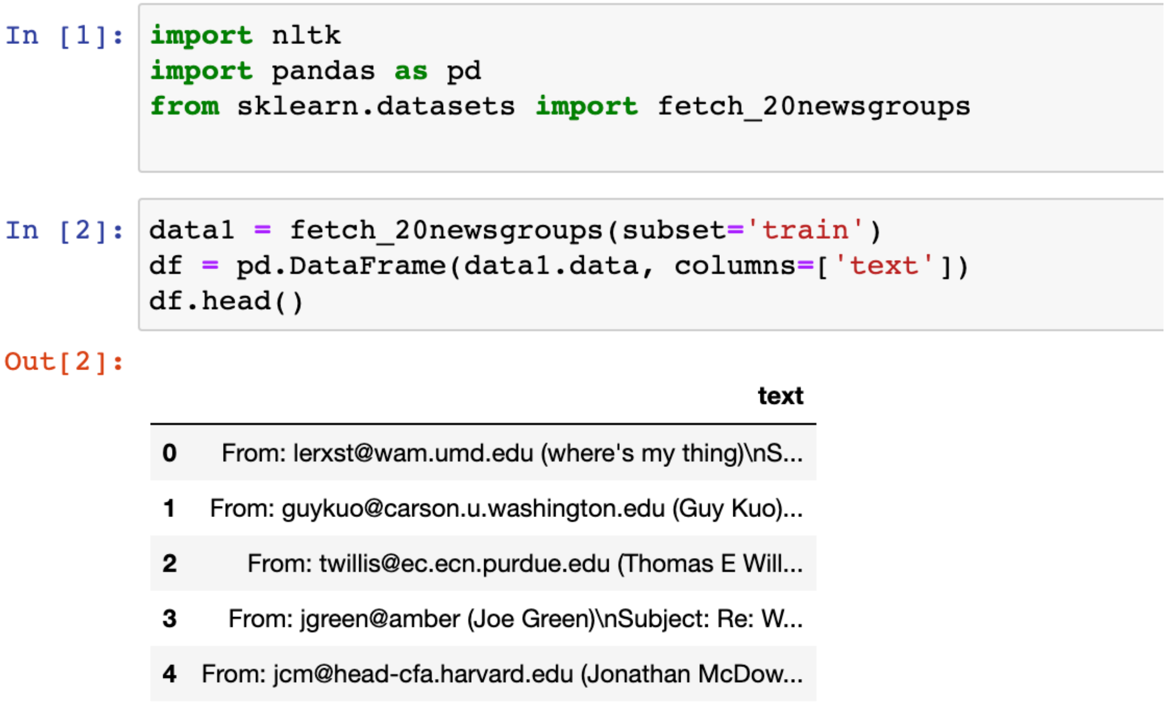
# Text Feature Engineering

- Here we discuss techniques that allow us to quickly extract features from short pieces of text, in the context of providing input to predictive models.
- Specifically, we will discuss how to capture text complexity by looking at some statistical parameters of the text such as the word length and count, the number of words and unique words used, the number of sentences, and so on.
- Our tools:
  - pandas
  - scikit-learn libraries
  - Natural Language Toolkit (NLTK)

# Text Feature Engineering

- We can capture text complexity by extracting the following information:
    - The total number of characters in the text
    - The total number of words
    - The total number of unique words
    - Lexical diversity = total number of words / number of unique words Word average length = number of characters / number of words

# Text Feature Engineering Example



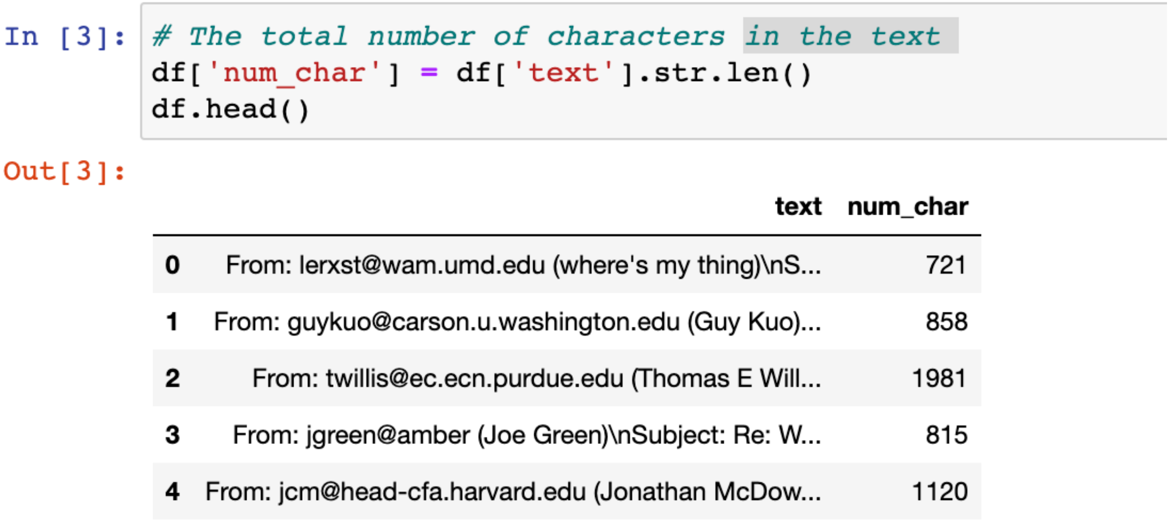
In [1]:
```
import nltk
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
```

In [2]:
```
data1 = fetch_20newsgroups(subset='train')
df = pd.DataFrame(data1.data, columns=['text'])
df.head()
```

Out[2]:

| | text |
|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... |

# Text Feature Engineering Example

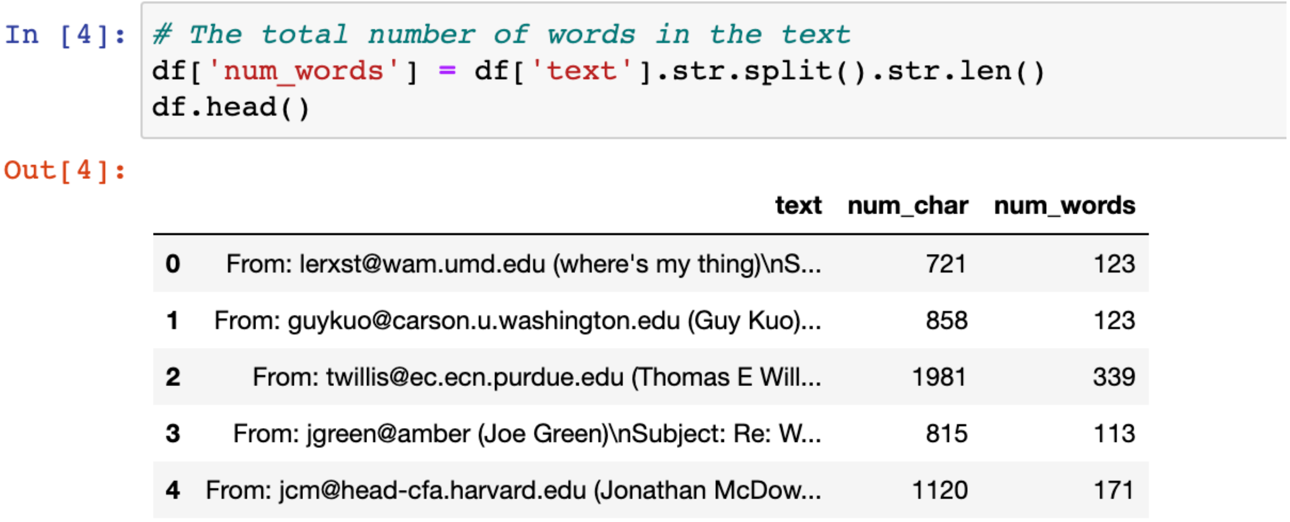


```
In [3]: # The total number of characters in the text
        df['num_char'] = df['text'].str.len()
        df.head()
```

Out[3]:

| | text | num_char |
|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 |

# Text Feature Engineering Example



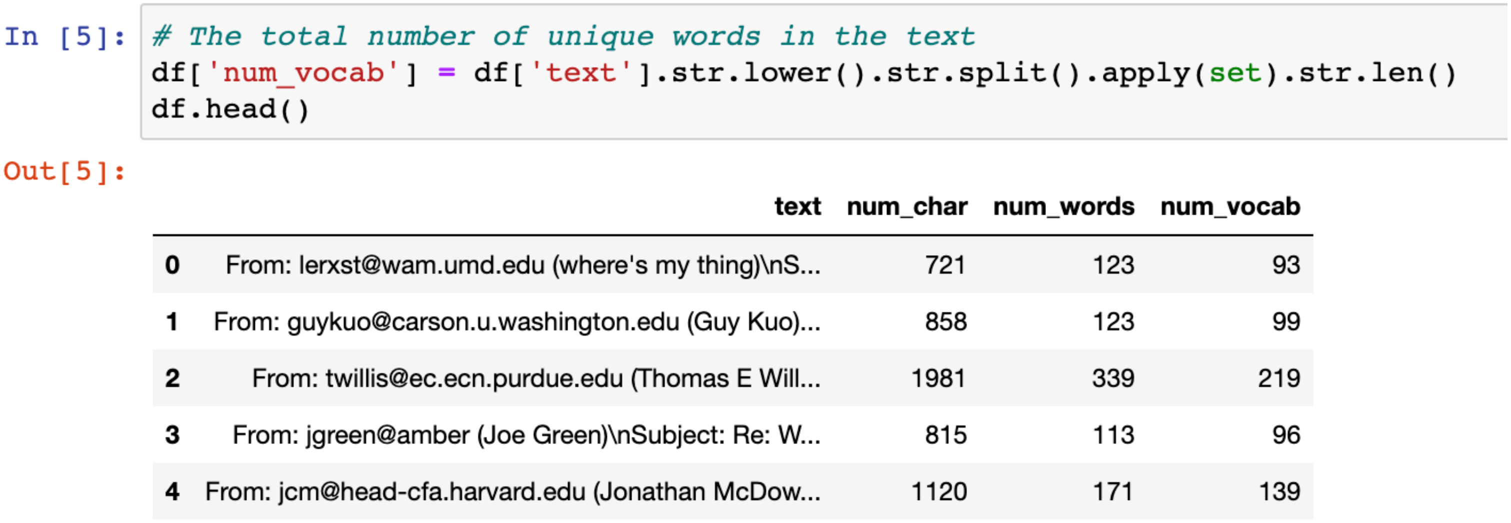
In [4]:

```python
# The total number of words in the text
df['num_words'] = df['text'].str.split().str.len()
df.head()
```

Out[4]:

| | text | num_char | num_words |
|---|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 | 123 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 | 123 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 | 339 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 | 113 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 | 171 |

# Text Feature Engineering Example



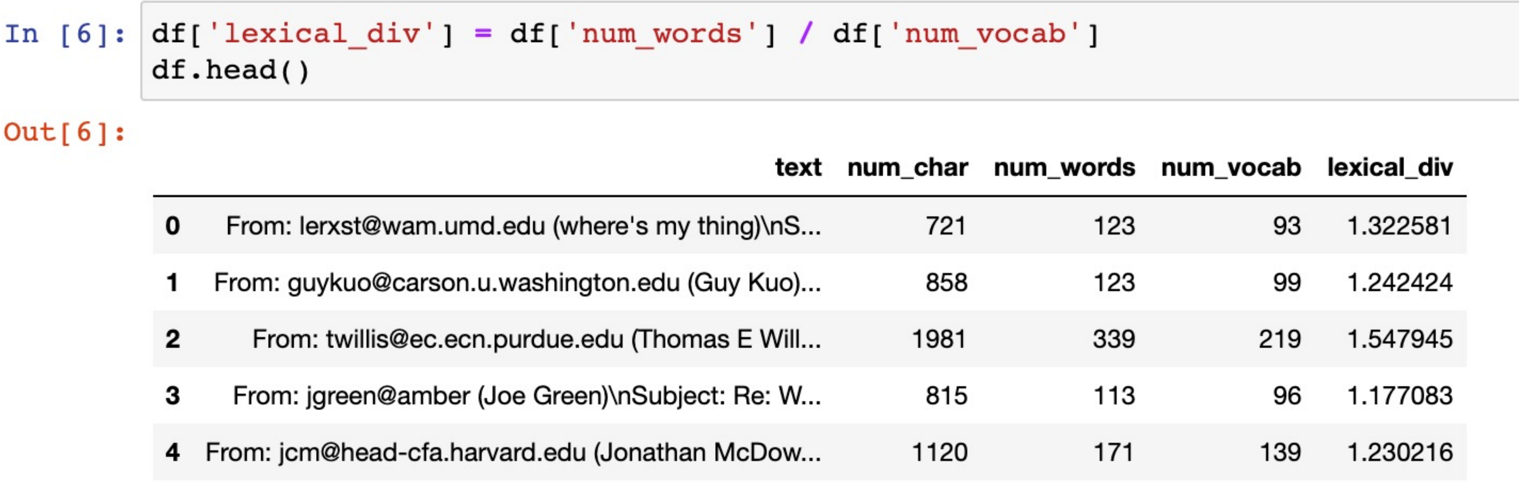
In [5]:

```
# The total number of unique words in the text
df['num_vocab'] = df['text'].str.lower().str.split().apply(set).str.len()
df.head()
```

Out[5]:

|  | text | num_char | num_words | num_vocab |
|---|---|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 | 123 | 93 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 | 123 | 99 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 | 339 | 219 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 | 113 | 96 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 | 171 | 139 |

# Text Feature Engineering Example



In [6]:

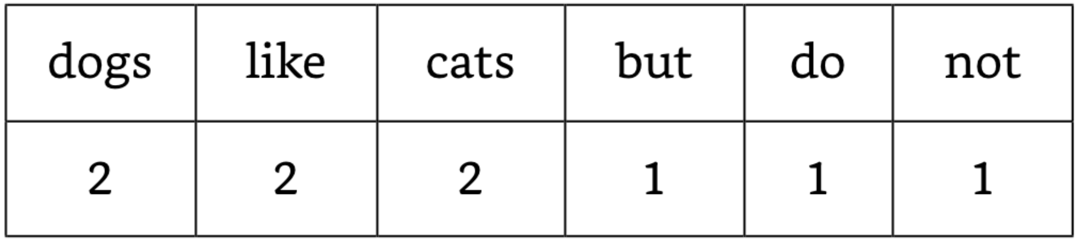
```
df['lexical_div'] = df['num_words'] / df['num_vocab']
df.head()
```

Out[6]:

| | text | num_char | num_words | num_vocab | lexical_div |
|---|---|---|---|---|---|
| **0** | From: lerxst@wam.umd.edu (where's my thing)\nS... | 721 | 123 | 93 | 1.322581 |
| **1** | From: guykuo@carson.u.washington.edu (Guy Kuo)... | 858 | 123 | 99 | 1.242424 |
| **2** | From: twillis@ec.ecn.purdue.edu (Thomas E Will... | 1981 | 339 | 219 | 1.547945 |
| **3** | From: jgreen@amber (Joe Green)\nSubject: Re: W... | 815 | 113 | 96 | 1.177083 |
| **4** | From: jcm@head-cfa.harvard.edu (Jonathan McDow... | 1120 | 171 | 139 | 1.230216 |

# Bag of Words (BoW)

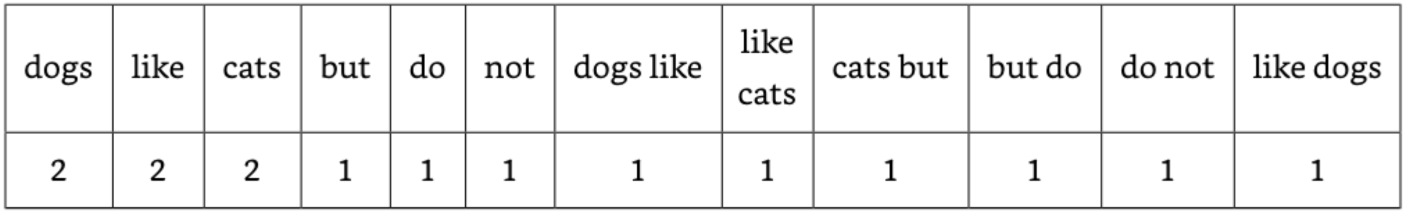
- A bag-of-words (BoW), is a simplified representation of a text that captures the words that are present in the text and the number of times each word appears in the text.
- Example: “Dogs like cats, but cats do not like dogs”



| dogs | like | cats | but | do | not |
|---|---|---|---|---|---|
| 2 | 2 | 2 | 1 | 1 | 1 |

# n-grams

- To capture some syntax, BoW can be used together with n-grams. An n-gram is a contiguous sequence of n items in a given text. Continuing with the sentence “Dogs like cats, but cats do not like dogs”, the derived 2-grams are as follows:
  - Dogs like
  - like cats
  - cats but
  - but do
  - do not
  - like dogs

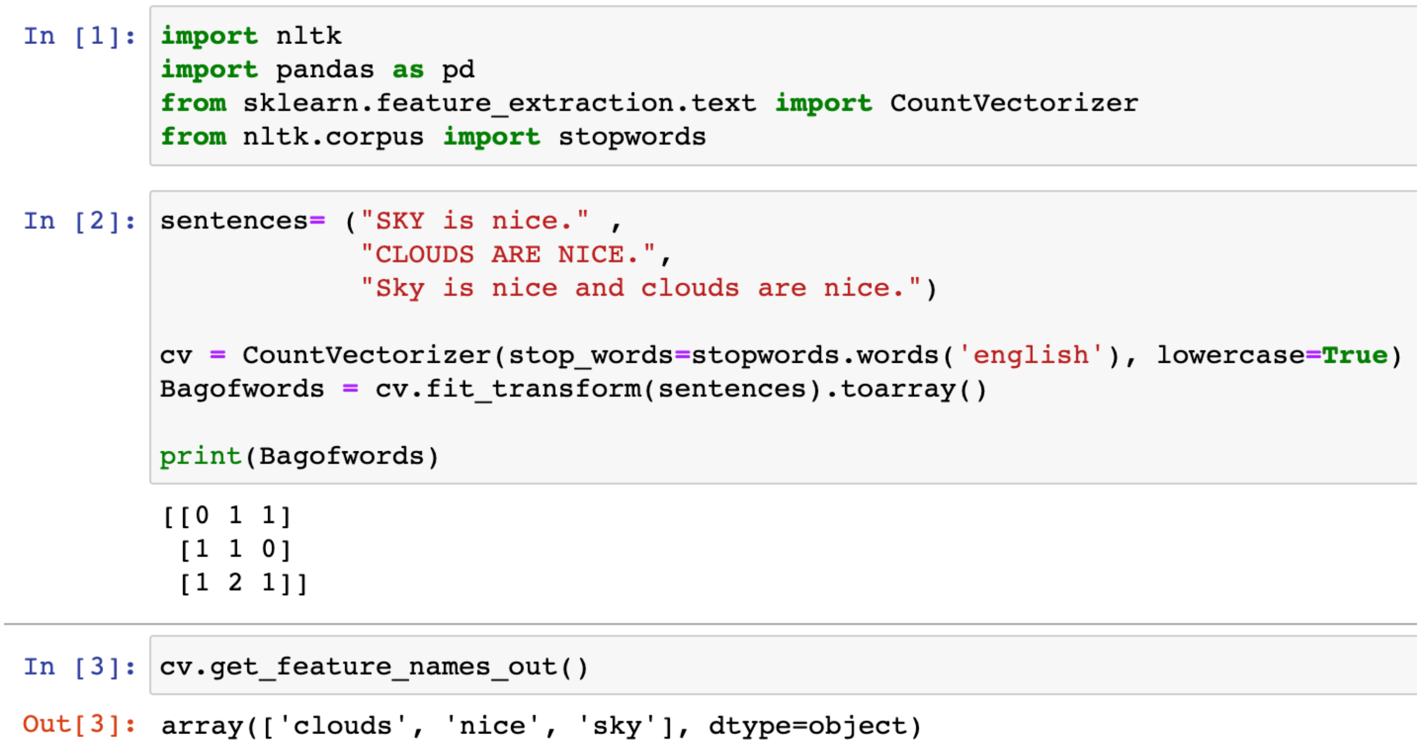


| dogs | like | cats | but | do | not | dogs like | like cats | cats but | but do | do not | like dogs |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# Bag of Words (BoW), n_grams

- CountVectorizer() is a transformer from scikit-learn which sets words in lowercase, removes stop words, retains words with a minimum acceptable frequency, and capture n-grams all together.

# Example BoW



In [1]:
```python
import nltk
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import stopwords
```

In [2]:
```python
sentences= ("SKY is nice." ,
            "CLOUDS ARE NICE.",
            "Sky is nice and clouds are nice.")

cv = CountVectorizer(stop_words=stopwords.words('english'), lowercase=True)
Bagofwords = cv.fit_transform(sentences).toarray()

print(Bagofwords)
```

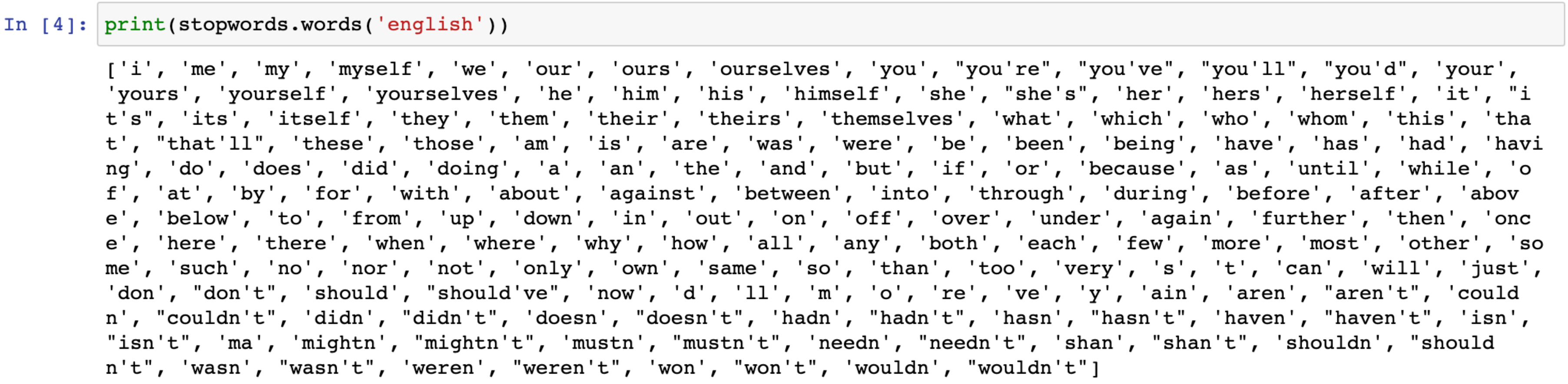
```
[[0 1 1]
 [1 1 0]
 [1 2 1]]
```

In [3]:
```python
cv.get_feature_names_out()
```

Out[3]:
```
array(['clouds', 'nice', 'sky'], dtype=object)
```

# Stop Words

Stop words are very frequently used words to make sentences flow, but that, per se, do not carry any useful information.



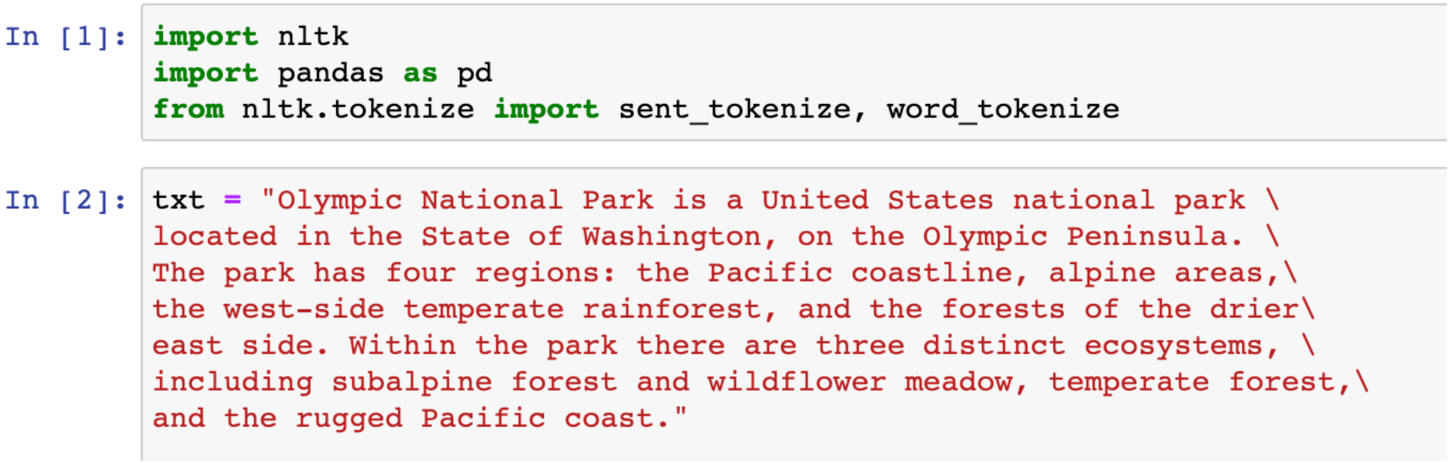
```
In [4]: print(stopwords.words('english'))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your',
'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "i
t's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'tha
t', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'havi
ng', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'o
f', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'abov
e', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'onc
e', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'so
me', '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', 'aren', "aren't", 'could
n', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven'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"]
```

# Tokenization

- Tokenization is used in NLP to split paragraphs and sentences into smaller units that can be more easily processed.
- The first step of the NLP process is gathering the data (text) and breaking it into understandable parts (sentences, words).

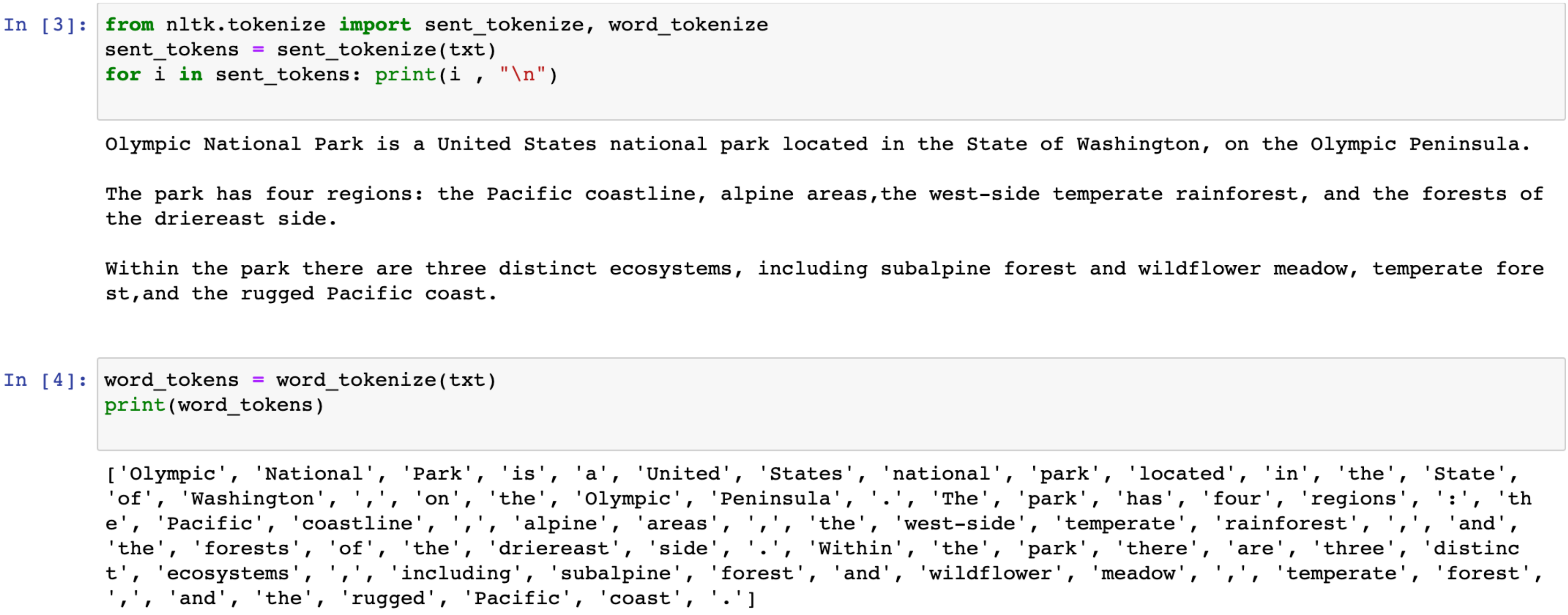
# Tokenization Example



```
import nltk
import pandas as pd
from nltk.tokenize import sent_tokenize, word_tokenize
```

```
txt = "Olympic National Park is a United States national park \
located in the State of Washington, on the Olympic Peninsula. \
The park has four regions: the Pacific coastline, alpine areas,\
the west-side temperate rainforest, and the forests of the drier\
east side. Within the park there are three distinct ecosystems, \
including subalpine forest and wildflower meadow, temperate forest,\
and the rugged Pacific coast."
```

# Tokenization Example



```
In [3]: from nltk.tokenize import sent_tokenize, word_tokenize
        sent_tokens = sent_tokenize(txt)
        for i in sent_tokens: print(i , "\n")
```

```
Olympic National Park is a United States national park located in the State of Washington, on the Olympic Peninsula.

The park has four regions: the Pacific coastline, alpine areas,the west-side temperate rainforest, and the forests of
the driereast side.

Within the park there are three distinct ecosystems, including subalpine forest and wildflower meadow, temperate fore
st,and the rugged Pacific coast.
```

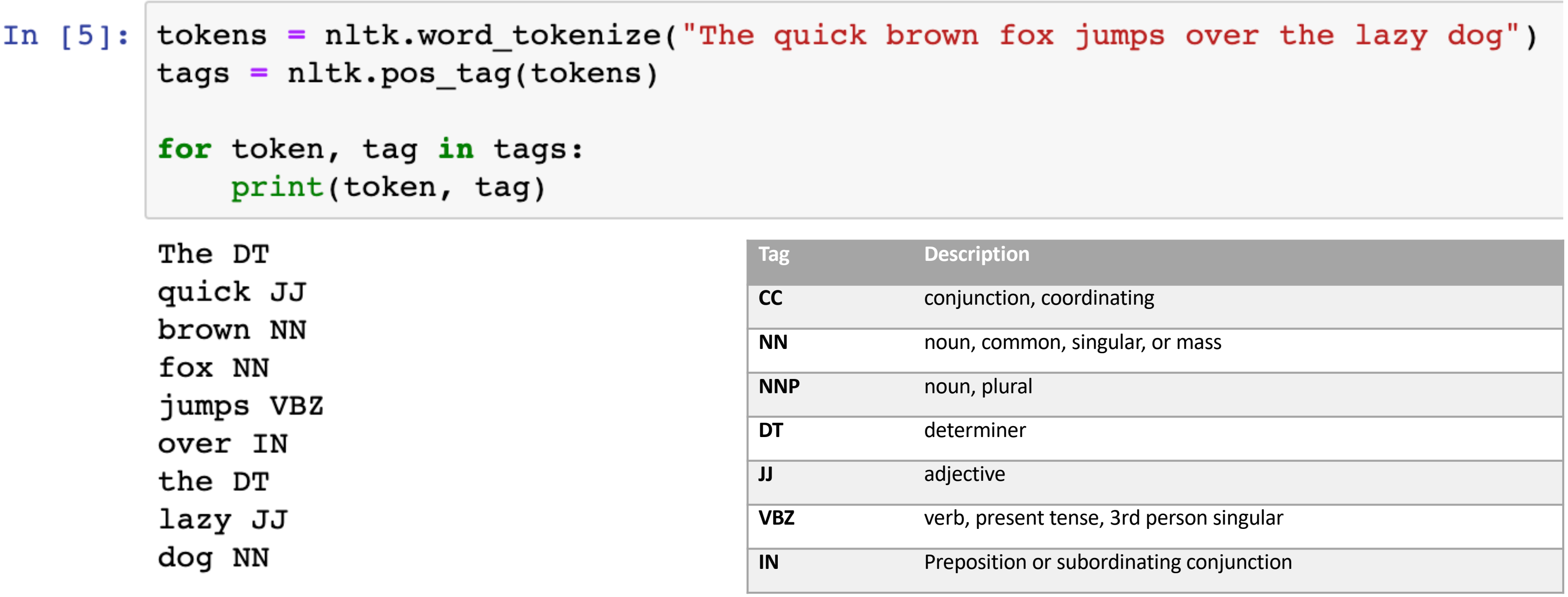
```
In [4]: word_tokens = word_tokenize(txt)
        print(word_tokens)
```

```
['Olympic', 'National', 'Park', 'is', 'a', 'United', 'States', 'national', 'park', 'located', 'in', 'the', 'State',
'of', 'Washington', ',', 'on', 'the', 'Olympic', 'Peninsula', '.', 'The', 'park', 'has', 'four', 'regions', ':', 'th
e', 'Pacific', 'coastline', ',', 'alpine', 'areas', ',', 'the', 'west-side', 'temperate', 'rainforest', ',', 'and',
'the', 'forests', 'of', 'the', 'driereast', 'side', '.', 'Within', 'the', 'park', 'there', 'are', 'three', 'distinc
t', 'ecosystems', ',', 'including', 'subalpine', 'forest', 'and', 'wildflower', 'meadow', ',', 'temperate', 'forest',
',', 'and', 'the', 'rugged', 'Pacific', 'coast', '.']
```

# (Part of Speech) PoS Tagging

- Part of speech tagging is an NLP technique that assigns a part of speech tag to each word in a sentence. The part of speech tag indicates the grammatical category of the word, such as noun, verb, adjective, adverb, etc.

# PoS Tagging Example



```
In [5]: tokens = nltk.word_tokenize("The quick brown fox jumps over the lazy dog")
        tags = nltk.pos_tag(tokens)

        for token, tag in tags:
            print(token, tag)
```

```
The DT
quick JJ
brown NN
fox NN
jumps VBZ
over IN
the DT
lazy JJ
dog NN
```

| Tag | Description |
|---|---|
| **CC** | conjunction, coordinating |
| **NN** | noun, common, singular, or mass |
| **NNP** | noun, plural |
| **DT** | determiner |
| **JJ** | adjective |
| **VBZ** | verb, present tense, 3rd person singular |
| **IN** | Preposition or subordinating conjunction |

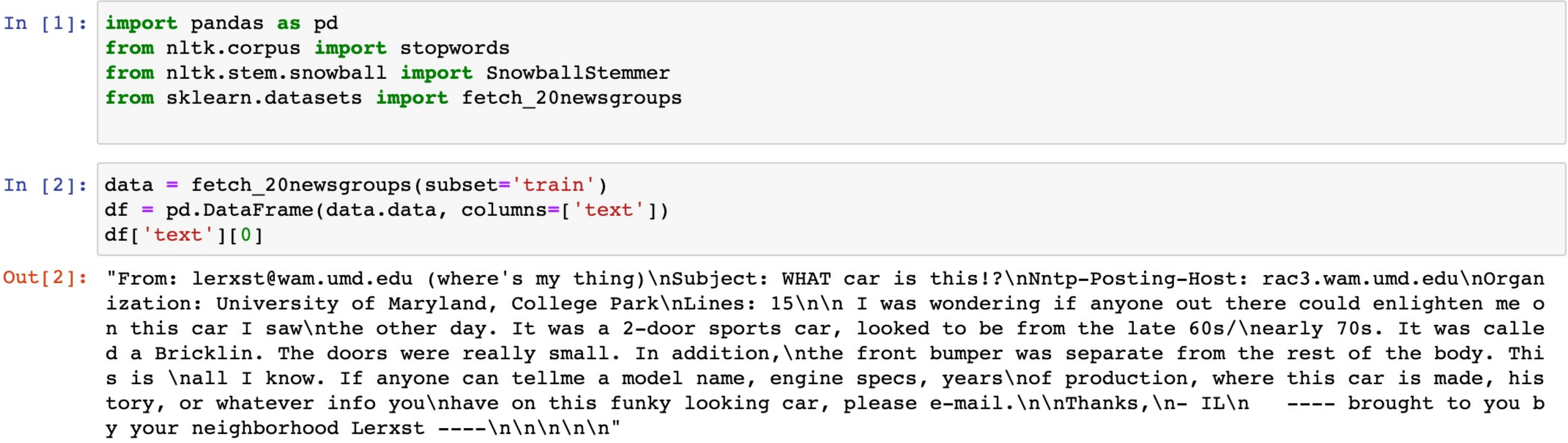
# Text Preprocessing Steps

- Text cleaning or text preprocessing involves:
  - Punctuation removal
  - Elimination of stop words
  - Character case setting
  - Word stemming

# Word Stemming

- Word stemming refers to reducing each word to its root or base so that the words playing, plays, and played become play, which, in essence, convey the same or very similar meaning.
    - had —> have
    - bananas —> banana
    - swimming —> swim

# Text Preprocessing Example

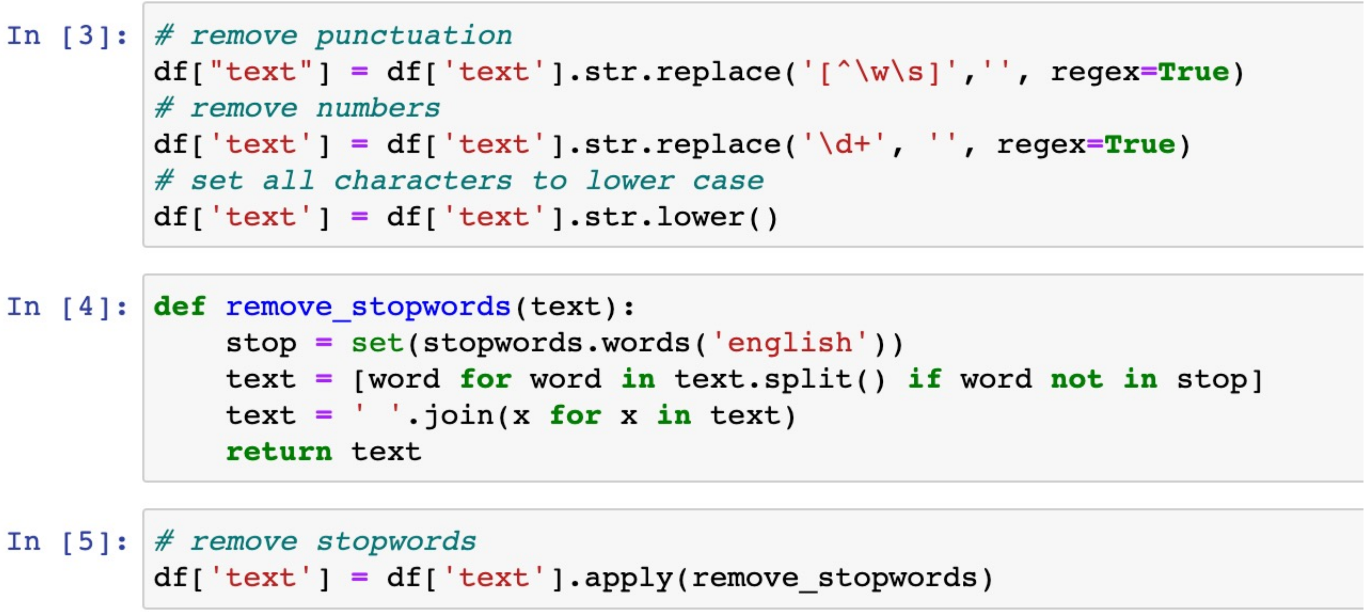


```
In [1]: import pandas as pd
        from nltk.corpus import stopwords
        from nltk.stem.snowball import SnowballStemmer
        from sklearn.datasets import fetch_20newsgroups
```

```
In [2]: data = fetch_20newsgroups(subset='train')
        df = pd.DataFrame(data.data, columns=['text'])
        df['text'][0]
```

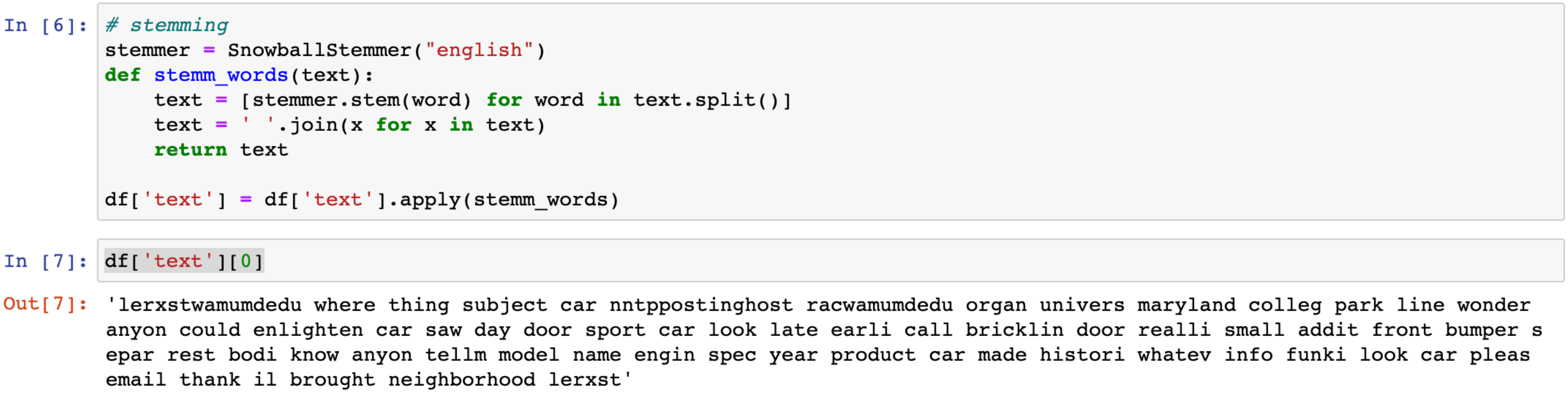
```
Out[2]: "From: lerxst@wam.umd.edu (where's my thing)\nSubject: WHAT car is this!?\nNntp-Posting-Host: rac3.wam.umd.edu\nOrgan
        ization: University of Maryland, College Park\nLines: 15\n\n I was wondering if anyone out there could enlighten me o
        n this car I saw\nthe other day. It was a 2-door sports car, looked to be from the late 60s/\nearly 70s. It was calle
        d a Bricklin. The doors were really small. In addition,\nthe front bumper was separate from the rest of the body. Thi
        s is \nall I know. If anyone can tellme a model name, engine specs, years\nof production, where this car is made, his
        tory, or whatever info you\nhave on this funky looking car, please e-mail.\n\nThanks,\n- IL\n   ---- brought to you b
        y your neighborhood Lerxst ----\n\n\n\n\n"
```

# Text Preprocessing Example



```
In [3]: # remove punctuation
        df["text"] = df['text'].str.replace('[^\w\s]','', regex=True)
        # remove numbers
        df['text'] = df['text'].str.replace('\d+', '', regex=True)
        # set all characters to lower case
        df['text'] = df['text'].str.lower()

In [4]: def remove_stopwords(text):
            stop = set(stopwords.words('english'))
            text = [word for word in text.split() if word not in stop]
            text = ' '.join(x for x in text)
            return text

In [5]: # remove stopwords
        df['text'] = df['text'].apply(remove_stopwords)
```

# Text Preprocessing Example



```
In [6]: # stemming
        stemmer = SnowballStemmer("english")
        def stemm_words(text):
            text = [stemmer.stem(word) for word in text.split()]
            text = ' '.join(x for x in text)
            return text

        df['text'] = df['text'].apply(stemm_words)
```

```
In [7]: df['text'][0]
```

```
Out[7]: 'lerxstwamumdedu where thing subject car nntppostinghost racwamumdedu organ univers maryland colleg park line wonder
        anyon could enlighten car saw day door sport car look late earli call bricklin door realli small addit front bumper s
        epar rest bodi know anyon tellm model name engin spec year product car made histori whatev info funki look car pleas
        email thank il brought neighborhood lerxst'
```

# Text Preprocessing

- The cleaning steps performed, resulted in strings containing the original text, without punctuation or numbers, in lowercase, without common words, and with the root of the word instead of its conjugated form.
- The data, as it is returned, can be used to derive features as described in the counting characters, words, and vocabulary or to create BoWs and TI-IDF matrices.

# Term Frequency-Inverse Document Frequency (TF-IDF)

- TF-IDF is a numerical statistic that captures how relevant a word is in a document, with respect to the entire collection of documents.
- Some words will appear a lot within a text document as well as across documents, for example, the English words the, a, and is. These words generally convey little information about the actual content of the document and don't make it stand out of the crowd.
- TF-IDF provides a way to weigh the importance of a word, by examining how many times it appears in a document, with respect to how often it appears across documents.
- Hence, commonly occurring words such as the, a, and is will have a low weight, and words more specific to a topic, such as leopard, will have a higher weight.

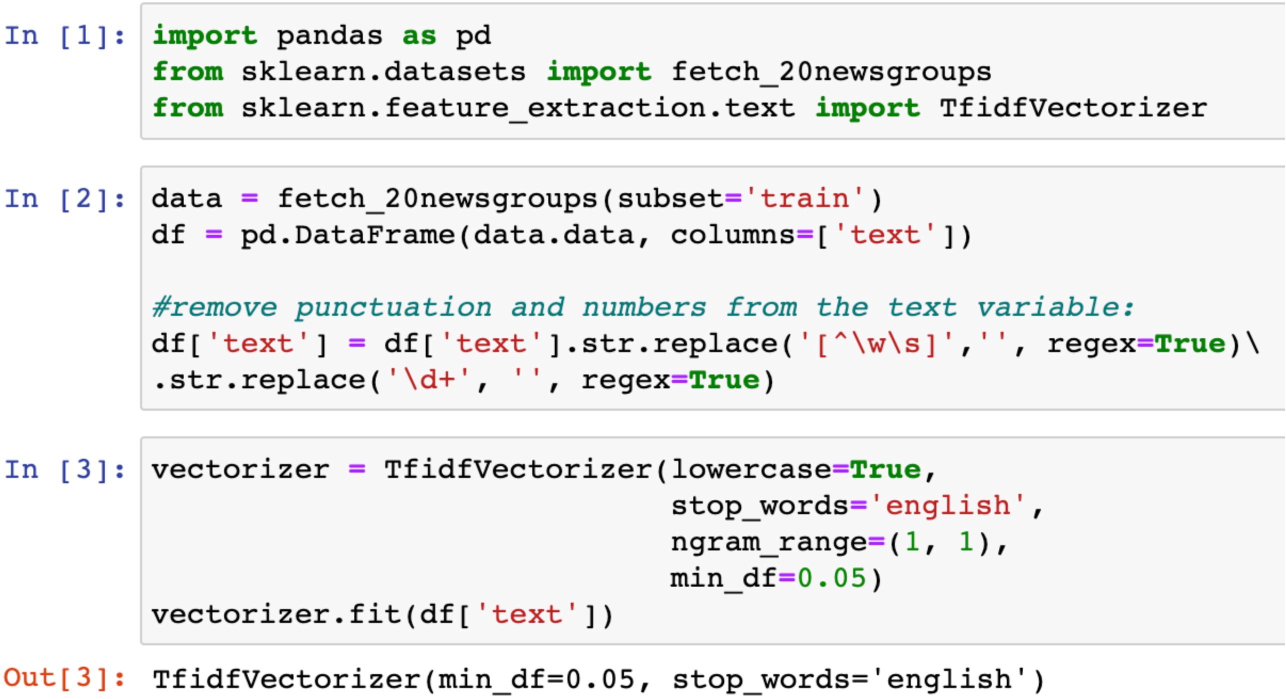
# TF-IDF

- TF-IDF = term frequency * inverse document frequency
- Term frequency is, the count of the word in an individual text. So, for term t, the term frequency is calculated as tf(t) = count(t) and is determined text by text.
- The inverse document frequency is a measure of how common the word is across all documents and is usually calculated on a logarithmic scale.

$$idf(t) = log(\frac{n}{1 + df(t)})$$

- Here, n is the total number of documents and df(t) the number of documents in which the term t appears.

# TF-IDF Example

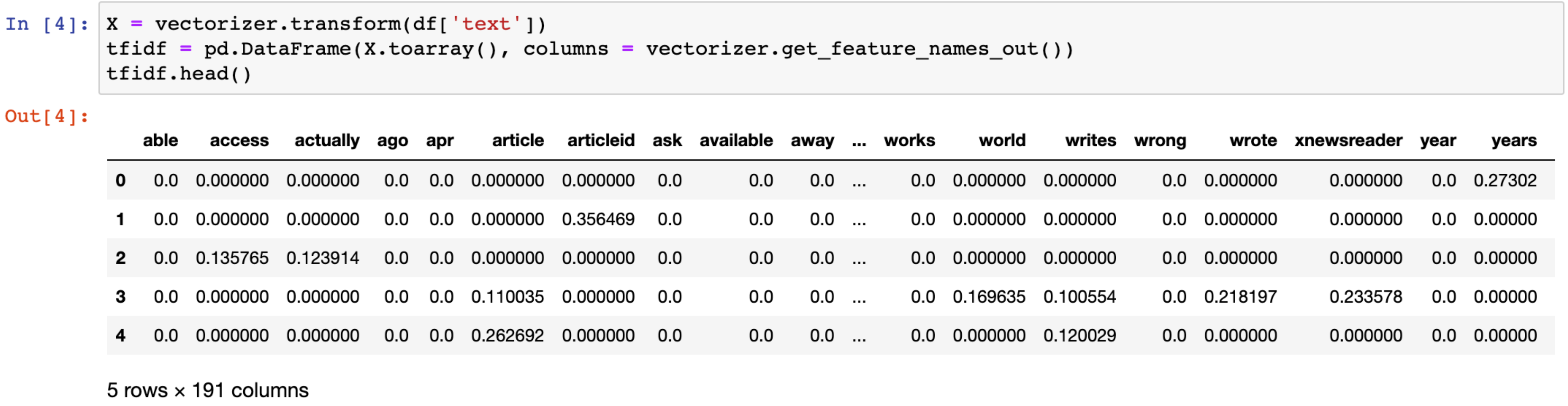


```
In [1]: import pandas as pd
        from sklearn.datasets import fetch_20newsgroups
        from sklearn.feature_extraction.text import TfidfVectorizer
```

```
In [2]: data = fetch_20newsgroups(subset='train')
        df = pd.DataFrame(data.data, columns=['text'])

        #remove punctuation and numbers from the text variable:
        df['text'] = df['text'].str.replace('[^\w\s]','', regex=True)\
        .str.replace('\d+', '', regex=True)
```

```
In [3]: vectorizer = TfidfVectorizer(lowercase=True,
                                     stop_words='english',
                                     ngram_range=(1, 1),
                                     min_df=0.05)
        vectorizer.fit(df['text'])

Out[3]: TfidfVectorizer(min_df=0.05, stop_words='english')
```

# TF-IDF Example



In [4]:
```
X = vectorizer.transform(df['text'])
tfidf = pd.DataFrame(X.toarray(), columns = vectorizer.get_feature_names_out())
tfidf.head()
```

Out[4]:

| | able | access | actually | ago | apr | article | articleid | ask | available | away | ... | works | world | writes | wrong | wrote | xnewsreader | year | years |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.27302 |
| **1** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.000000 | 0.356469 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.00000 |
| **2** | 0.0 | 0.135765 | 0.123914 | 0.0 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.00000 |
| **3** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.110035 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.169635 | 0.100554 | 0.0 | 0.218197 | 0.233578 | 0.0 | 0.00000 |
| **4** | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.0 | 0.262692 | 0.000000 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.000000 | 0.120029 | 0.0 | 0.000000 | 0.000000 | 0.0 | 0.00000 |

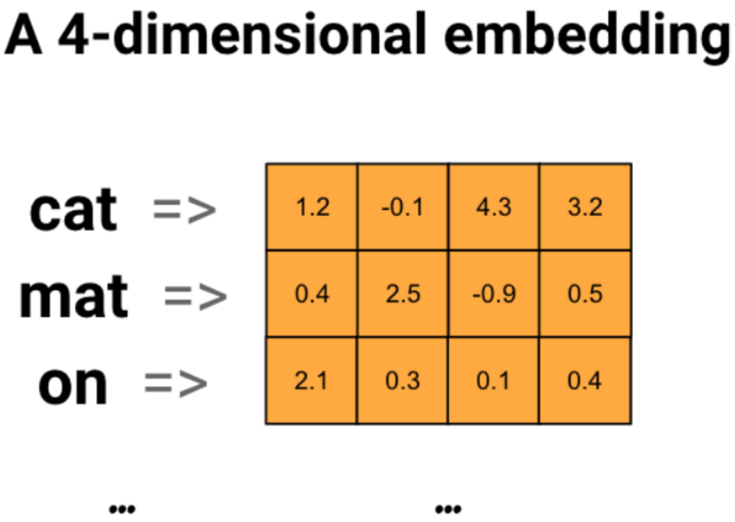
5 rows × 191 columns

# Word Embedding

- Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding.
- Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer).
- It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets.

# Word Embedding

- Each word is represented as a 4-dimensional vector of floating point values.
- Another way to think of an embedding is as "lookup table".
- After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.



A 4-dimensional embedding

| | | | | | |
|---|---|---|---|---|---|
| cat | => | 1.2 | -0.1 | 4.3 | 3.2 |
| mat | => | 0.4 | 2.5 | -0.9 | 0.5 |
| on | => | 2.1 | 0.3 | 0.1 | 0.4 |
| ... | | ... | | | |

# Popular Word Embedding Methods

- Word2Vec
- GLoVe
- FastText

# Word2Vec

- Word2Vec uses a neural network used to convert a given text corpus into a vector of words. The vector which represents the word is called Neural Word Embeddings.
- Continuous bag-of-words (CBOW): predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.
- Continuous skip-gram: predicts words within a certain range before and after the current word in the same sentence.

# CBOW & Skip-Gram

- Assume we give the input (context) ’Today is … and the weather is also very good‘, CBOW can predict the missing word is ’amazing‘.
- The missing word is selected from a set of words (e.g. {amazing, probability =0.8}, {snowy, probability = 0.03}, …) and each of them has a probability, the word ”amazing“ has the highest probability and thus it is selected by CBOW to be returned as the predicted word.

- Skip-Gram is reverse to CBOW. It finds the context words for the given target word.
- For example, giving the context word ”cow“ the skip-gram can find the surrounding words such as ’heavy‘, ’white’, etc. Because there are sentences in the text corpus such as: ’The big white cow is heavy to move‘. Skip-gram uses those sentences to identify surrounding words for ’cow.

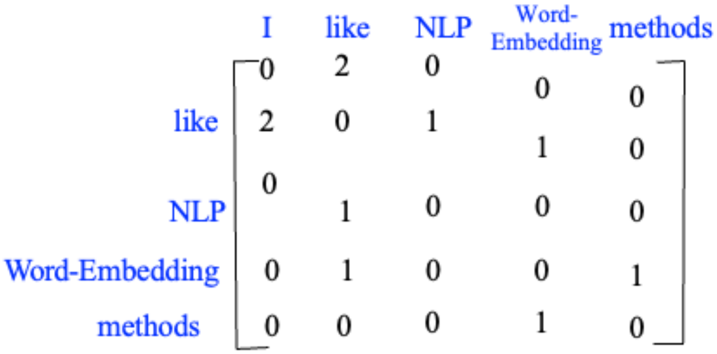
# GloVe

- The abbreviation of GloVe is for Global Vector for Word Representation, and it is introduced in 2014 [Pennington ’14] one year after word2Vec, to address a Word2Vec co-occurrence limitation.
- Word2Vec goes through each word in the entire corpus and predicts the surrounding words once for every word.
- In particular, Word2Vec‘s skip-gram ignores whether some context words appear more often than others. A frequent co-occurrence of words creates more instances for the training set, but those instances do not have any additional information; they only increase the size of the training dataset.
- GloVe resolves this challenge by going through the entire corpus once and counting the co-occurrences once for all words. The result will be a matrix that involves all co-occurrences, i.e. window-based co-occurrence matrix.

# GloVe

- For example, consider two following sentences and their simplified symmetric *co-occurrence matrix*, which we only check one neighbor words (window size=1). Based on the existence of a window word it adds one point into the matrix. Since, ’I‘ is neighbor to ’like‘ two times, their value in the matrix will be 2.

I like NLP  
I like Word-Embedding methods



$$\begin{array}{c|ccccc|} & \text{I} & \text{like} & \text{NLP} & \text{Word-Embedding} & \text{methods} \\ & 0 & 2 & 0 & 0 & 0 \\ \text{like} & 2 & 0 & 1 & 1 & 0 \\ \text{NLP} & 0 & 1 & 0 & 0 & 0 \\ \text{Word-Embedding} & 0 & 1 & 0 & 0 & 1 \\ \text{methods} & 0 & 0 & 0 & 1 & 0 \end{array}$$

# FastText

- Both Word2Vec and GloVe have the problem of generalization to unknown words.
- Skip-gram of Word2Vec uses uni-gram (one complete word) and takes every single word as a single unit, but FastText uses n-gram and considers n number of characters together as a single unit.
- In simple words, while using FastText, we have a vector for a single word instead of having a vector for a set of adjacent words.
- For example, assuming a window size of three, in the training set the word ’aquarium‘ can have one vector in word2Vec, but in FastText we will have ’aqu‘, ’qua‘, ’uar‘, ’ari‘, and ’ium‘. Now if the test set we encounter the word ’aqua‘ for the first time, FastText can see that it is like ’aquarium‘, because the vector values of ’aqu’ and ’qua’ same

# Difference between non-contextual and contextual word embeddings

- Word2Vec, as a word embedding technique, is context-independent. Each word is represented by a fixed, context-free vector in the embedding space. The context-independent nature of Word2Vec means that the same word will have the same vector representation regardless of its context.
- In word embedding methods words with multiple meanings are conflated into a single representation.

# From Word Embedding to Context Understanding

- I went to the bank …
- I went to the bank of the river.

# Contextual Embedding Methods

- Contextual embeddings are a method of assigning a representation to each word based on its context. They are used to learn sequence-level semantics by considering the sequence of all words in a document.
- Contextual embeddings capture word semantics in context. They can represent the same word differently depending on the context. For example, "left" can have different meanings depending on the context.
- Contextual embeddings assign a vector to each "token". The model can predict the next word given the current input word.

# Advances After Word Embeddings

- Seq2Seq Neural Networks (2014)
- Seq2Seq with Attention (2014, 2015)
- Transformer architecture (2017)
- BERT (2018)
- After BERT a competition to make a bigger and thus more accurate model (e.g. GPT-3, LLama, Bloom, ….)

# Transformer (Machine Learning)

- A transformer is a deep learning architecture that relies on the parallel multi-head attention mechanism.
- This technique is used for training large language models on large (language) datasets, such as the Wikipedia corpus and Common Crawl.
- Transformer models are made up of an encoder and a decoder. They are pre-trained on a large corpus of text data and can be fine-tuned for specific tasks.
- Examples of transformer models: GPT-2, GPT-3, GPT-4, Claude, BERT, XLNet, RoBERTa, ChatGPT, BioGPT, CodeGen, LLaMa, BART, BlenderBot 3, BLOOM.

# Language Models

- A language model is a probabilistic model of a natural language that can generate probabilities of a series of words, based on text corpora in one or multiple languages it was trained on.
- Language modeling is unsupervised. Language models are multitasking learners that are used for natural language processing (NLP) tasks like:
  - Audio to text conversion
  - Speech recognition
  - Sentiment analysis
  - Summarization

# Sentiment Analysis

# What is Sentiment Analysis

- Sentiment analysis is used for analyzing text and determining whether it conveys a positive, negative, or neutral sentiment.
- Sentiment analysis is the process to automatically enabling machines to understand opinions about something.
- With Sentiment analysis, we can automate the product review. e.g., “Sentimental Analysis for Airline on Twitter data”.

# Sentiment Analysis – Basic Consepts

- The term sentiment analysis is also referred to as opinion mining. It has revolutionized the field of NLP (Natural Language Processing).
- In sentiment analysis, three types of information are required to get identified automatically:
  - Polarity: The positiveness/negativeness of the speaker's tone
  - Subject: The subject of the sentence or the topic being discussed.
  - Opinion holder: The subject who is expressing their feeling.

# Opinion

- Text information can be categorized into fact and opinion.

  Facts are objective expression about a topic.

  Opinions are subjective expression about a topic.
- Subjective example: I don’t like traveling by train.
- Objective example: Trains are means of transportation.
- Classifying a sentence as a subjective or objective sentence is known as subjectivity classification.
- Classifying a sentence as expressing a positive, neutral or negative opinion is known as polarity classification.

# Types of Opinion

We have two types of opinions: direct and comparative.

1. Its’ storage is too small. <— direct
2. Its storage is smaller than product X. <— comparative

Usually, comparative opinion includes a comparison. In our example product X received positive opinion and the other one which is the topic of discussion receives negative opinion.

# Types of Opinion

Explicit versus Implicit

The picture quality is good. <— **Explicit (**about the picture)

Its contrast is different than the other one. <— **Implicit (**about the picture)

# Sentiment Analysis

Sentiment Analysis is usually focused on one of the three components:

1. Polarity (positive, neutral, negative), which is also called fine grained sentiment analysis.

2. Emotions (Happy, Sad, Angry, …). Lexicons are not good at detecting emotions, and usually, these research works focus on facial expression, except for analyzing the text.

3. Intentions (interested, not interested)

# Intend Analysis

- In contrast to sentiment analysis, which focuses on user emotions and sentiments. Intend analysis, is trying to understand what are subjects willing to do.
- Intent Analysis involves understanding the emotions and intent of a user.
- The intent analyzer tries to identify users' intent and deliver content related to the users intend.
- Example

I am moving to Boston to advance my career.

# Sentiment Analysis Applications

- Social Media Analysis: using X to predict election result and analysis people's concern.
- Product or Brand Monitoring and Market Research Analysis
- Consumer Market Analysis
- Public Health Analysis

# Advantages of Sentiment Analysis

- Scalability: Manually collecting customers data, market trends, … is cumbersome and expensive process.

- Real-time Analysis: There could be a crisis happening at some points and requires an intervention from supervisors. The real-time capability of sentiment analysis can identify those crisis. For example, a very angry customer is chatting with tech support.

- Consistency: Humans are not very good in quantifying sentiments, by using machines the subjectivity of quantification could be mitigated.

# Advantages of Sentiment Analysis

- Scalability: Manually collecting customers data, market trends, … is cumbersome and expensive process.

- Real-time Analysis: There could be a crisis happening at some points and requires an intervention from supervisors. The real-time capability of sentiment analysis can identify those crisis. For example, a very angry customer is chatting with tech support.

- Consistency: Humans are not very good in quantifying sentiments, by using machines the subjectivity of quantification could be mitigated.

# Sentiment Analysis Approach

- Dictionary Based
- Machine Learning (ML)
- Hybrid (combine both rule based and ML approach to achieve a higher accuracy)

# Dictionary Based Approach

- The developer manually creates rules and creates a sentiment analysis algorithm based on these rules.
- The algorithm uses human created rules and performs the sentiment analysis.
- Usually, rules are received one of the following inputs:
  - Classical text features, tokenization, stemming,…
  - Lexicon resources, which list words and their expressions, such as LIWC.

Linguistic Inquiry and Word Count (LIWC) is a text analysis program that calculates the percentage of words in each text that fall into one or more of over 80 linguistic, psychological and topical categories indicating various social, cognitive, and affective processes.

# Sample algorithm for Dictionary based Sentiment Analysis

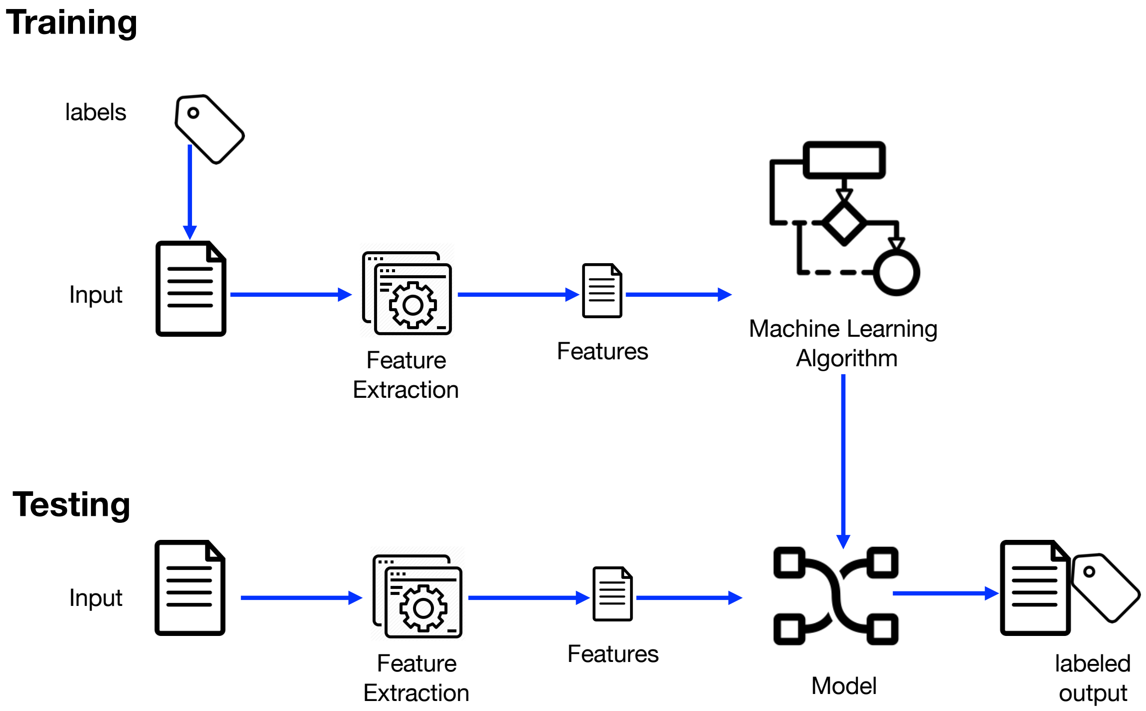
1. Receive (i) Input text, list of (ii) positive words and (ii) negative words.

2. Tokenize the input text.

3. Count its number of positive and negative words.

4. Based on the counted number of positive and negative words assign each token a sentiment score.

# Machine Learning based Sentiment Analysis



# Sentiment Analysis Challenges

**Subjectivity and lack of tone**

What is the sentiment of this: *Oh, another smartphone with big camera!*

**Lack of context-awareness**

What is the sentiment of this: … ? *Answer: Absolutely nothing.*

**Sarcasm**

What is the sentiment of this: *of course, I am happy*

# Sentiment Analysis Methods

The three general purpose baseline lexicons:

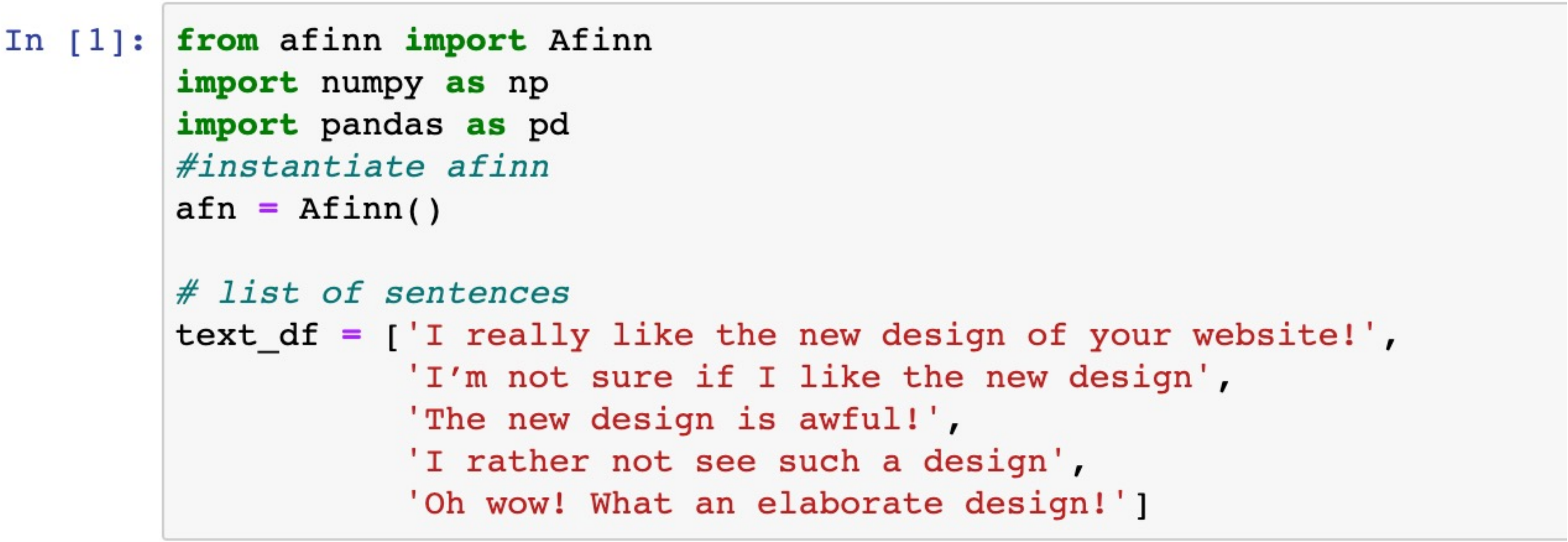
**NRC Emotion Lexicon** is a list of 5,636 English words and their associations with eight basic emotions and two sentiments. The NRC lexicon categorizes words in a binary fashion into categories of: Positive, Negative, Anger, Anticipation, Disgust, Fear, Joy, Sadness, Surprise and Trust.

**AFINN lexicon** developed by Finn Arup Nielsen, assigns words with a score that runs between -5 and 5. It contains 3300+ words with a polarity score associated with each word.

**Bing sentiment lexicon** developed by Bing Liu et al., categorizes words as 0/1 within a positive and negative category.

All these lexicons are based on unigram (single word), which had its limitation as well.

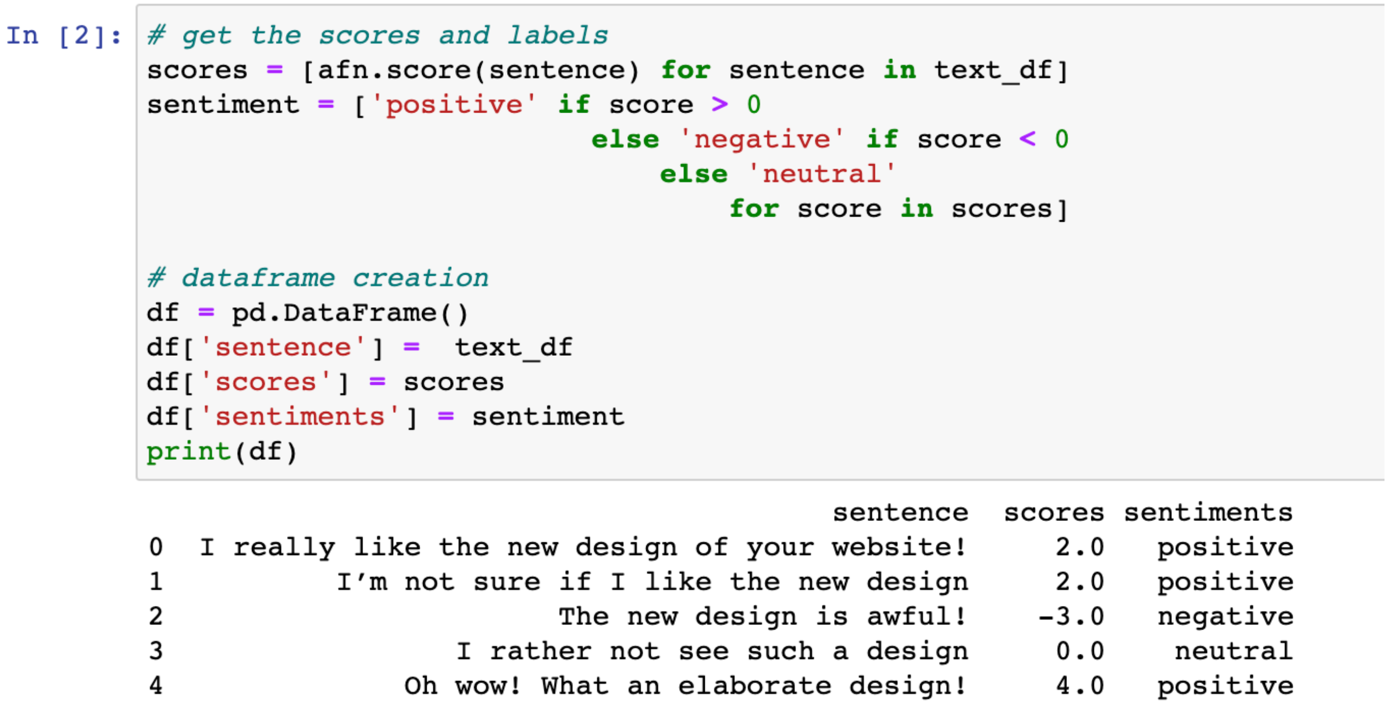
# Sentiment Analysis Example (AFFIN)



In [1]:

```
from afinn import Afinn
import numpy as np
import pandas as pd
#instantiate afinn
afn = Afinn()

# list of sentences
text_df = ['I really like the new design of your website!',
           'I’m not sure if I like the new design',
           'The new design is awful!',
           'I rather not see such a design',
           'Oh wow! What an elaborate design!']
```

# Sentiment Analysis Example (AFFIN)

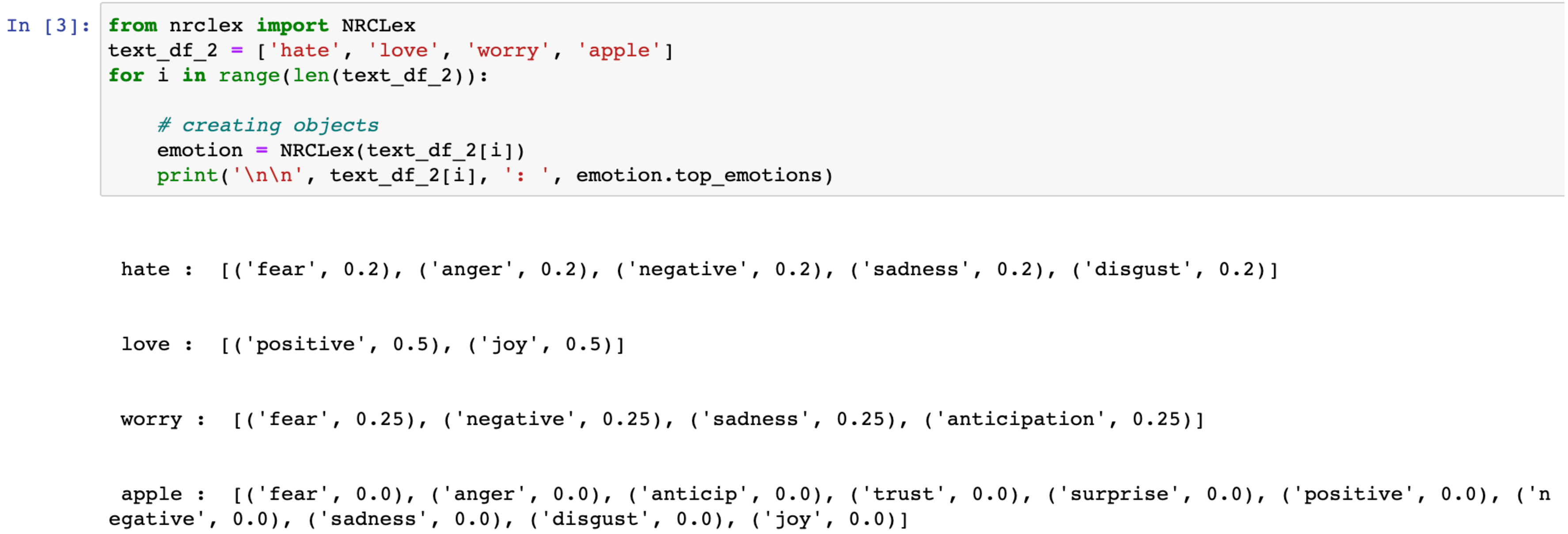


In [2]:

```
# get the scores and labels
scores = [afn.score(sentence) for sentence in text_df]
sentiment = ['positive' if score > 0
                          else 'negative' if score < 0
                              else 'neutral'
                                  for score in scores]

# dataframe creation
df = pd.DataFrame()
df['sentence'] =  text_df
df['scores'] = scores
df['sentiments'] = sentiment
print(df)
```

```
                                      sentence  scores sentiments
0  I really like the new design of your website!     2.0   positive
1          I’m not sure if I like the new design     2.0   positive
2                        The new design is awful!    -3.0   negative
3                  I rather not see such a design     0.0    neutral
4                Oh wow! What an elaborate design!     4.0   positive
```

# Sentiment Analysis Example (NRC)



```
In [3]: from nrclex import NRCLex
        text_df_2 = ['hate', 'love', 'worry', 'apple']
        for i in range(len(text_df_2)):

            # creating objects
            emotion = NRCLex(text_df_2[i])
            print('\n\n', text_df_2[i], ': ', emotion.top_emotions)
```

```
 hate :  [('fear', 0.2), ('anger', 0.2), ('negative', 0.2), ('sadness', 0.2), ('disgust', 0.2)]


 love :  [('positive', 0.5), ('joy', 0.5)]


 worry :  [('fear', 0.25), ('negative', 0.25), ('sadness', 0.25), ('anticipation', 0.25)]


 apple :  [('fear', 0.0), ('anger', 0.0), ('anticip', 0.0), ('trust', 0.0), ('surprise', 0.0), ('positive', 0.0), ('n
egative', 0.0), ('sadness', 0.0), ('disgust', 0.0), ('joy', 0.0)]
```

# References

- Illowsky, Barbara; Dean, Susan. Introductory Statistics. XanEdu Publishing Inc.
- Bruce, Peter; Bruce, Andrew; Gedeck, Peter. Practical Statistics for Data Scientists . O'Reilly Media.
- Müller, Andreas C.; Guido, Sarah. Introduction to Machine Learning with Python. O'Reilly Media. Kindle Edition.
- Galli, Soledad. Python Feature Engineering Cookbook: Over 70 recipes for creating, engineering, and transforming features to build machine learning models Packt Publishing. Kindle Edition.
- https://www.tensorflow.org/text/guide/word_embeddings
- "Thumbs Up? Sentiment Classification using Machine Learning Techniques" by Pang and Lee, published in 2002.