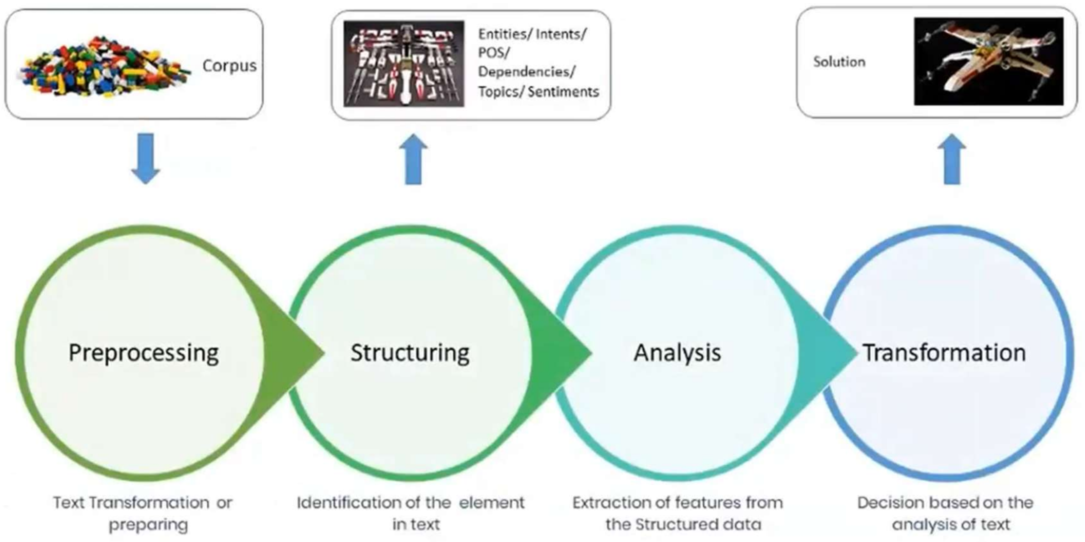
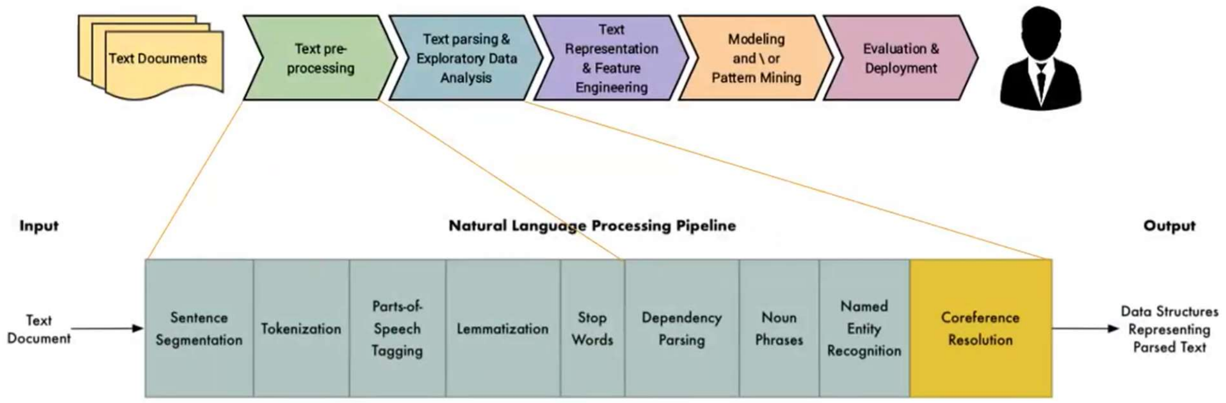
Divide Complex task to manageable pieces


Corpus
Entities/ Intents/ POS/ Dependencies/ Topics/ Sentiments
Solution
Preprocessing
Structuring
Analysis
Transformation
Text Transformation or preparing
Identification of the element in text
Extraction of features from the Structured data
Decision based on the analysis of text

Pipeline - Divide Complex task to manageable pieces



Tf-idf = Term frequency and **Inverse document frequency**.

TF-IDF scores provide weight to a word in proportion of the impact it has on the meaning of a sentence.

TF-IDF is product of 2 independent scores, term frequency(tf) and inverse document frequency (idf)

TFIDF

For a term i in document j:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

e.g. document with a length of 150 words

target term 'dog'

If 'dog' appeared in the document 5 times

$$tf_{weight} = \frac{\#\ of\ occurence\ of\ target\ term}{\#\ of\ total\ terms\ in\ the\ document} \qquad \frac{5}{150} = 0.033$$

Tf-idf = Term frequency and **Inverse document frequency**.

TF-IDF scores provide weight to a word in proportion of the impact it has on the meaning of a sentence.

TF-IDF is product of 2 independent scores, term frequency(tf) and inverse document frequency (idf)

TFIDF

For a term i in document j:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

idf weight measures the significance of the word in a corpus

if 'dog' appeared in 10,000 documents out of a 1 million-document dataset

$$idf_{weight} = \log\left(\frac{\#\ of\ documents\ in\ the\ dataset}{\#\ of\ documents\ containing\ the\ target\ term}\right) \qquad \log\left(\frac{1,000,000}{10,000}\right) = 2$$

Finally, the **tf-idf weigh**t of the term 'dog' is

$$tf\,idf_{weight} = tf_{weight} \cdot idf_{weight} \qquad 0.033 \cdot 2 = 0.067$$

Text Representation: Embeddings - Illustration

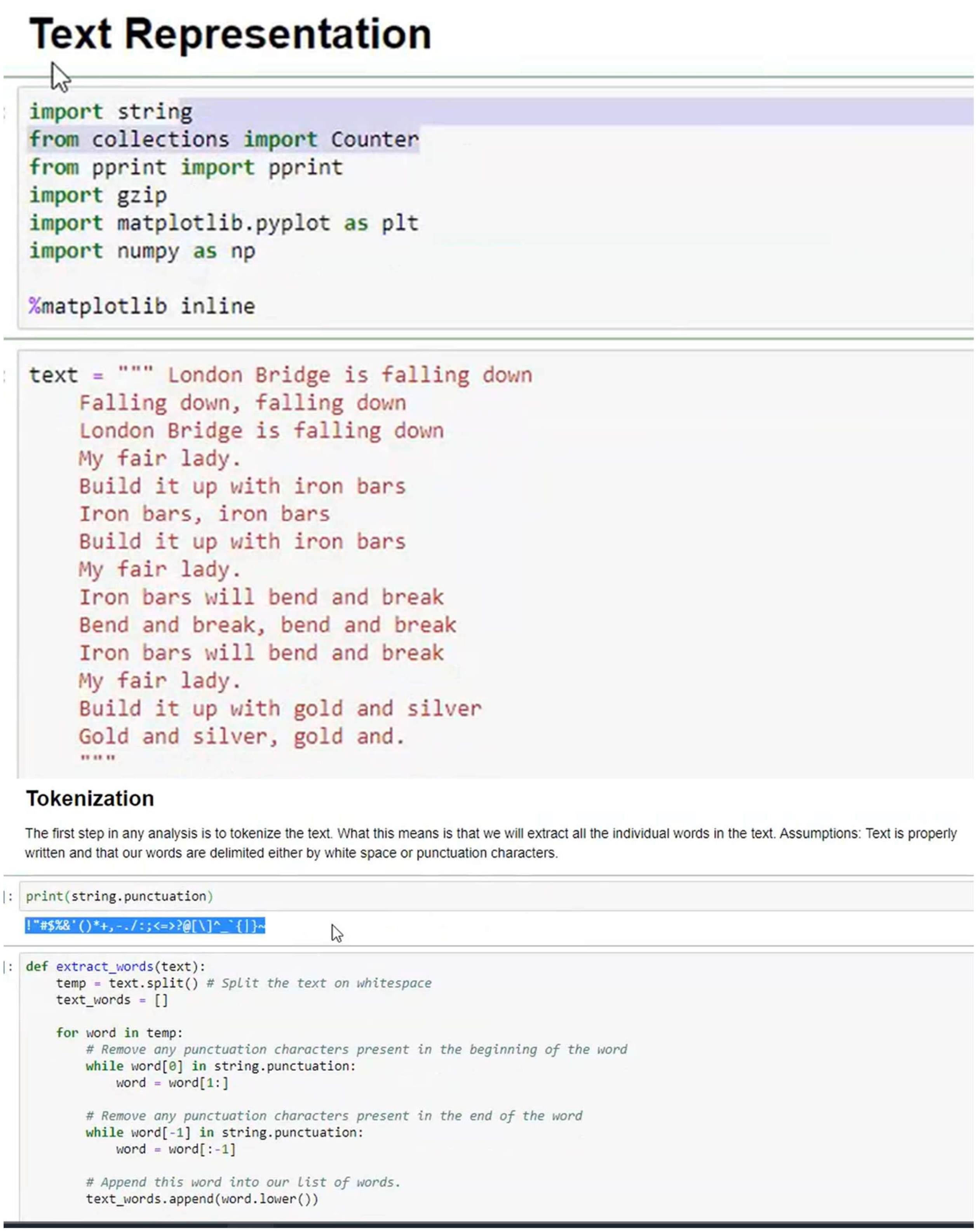
Two dimensions may not be enough to capture enough information about how different people

	Trait #1	Trait #2	Trait #3	Trait #4	Trait #5
Jay	-0.4	0.8	0.5	-0.2	0.3
Person #1	-0.3	0.2	0.3	-0.4	0.9
Person #2	-0.5	-0.4	-0.2	0.7	-0.1

```
cosine_similarity( Jay [-0.4 0.8 0.5 -0.2 0.3] , Person #1 [-0.3 0.2 0.3 -0.4 0.9] ) = 0.66 ✓

cosine_similarity( Jay [-0.4 0.8 0.5 -0.2 0.3] , Person #2 [-0.5 -0.4 -0.2 0.7 -0.1] ) = -0.37
```

cosine similarity works for any number of dimensions.

The problem with more dimensions is that it becomes difficult to represent in two dimensions.



Text Representation

```python
import string
from collections import Counter
from pprint import pprint
import gzip
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

```python
text = """ London Bridge is falling down
    Falling down, falling down
    London Bridge is falling down
    My fair lady.
    Build it up with iron bars
    Iron bars, iron bars
    Build it up with iron bars
    My fair lady.
    Iron bars will bend and break
    Bend and break, bend and break
    Iron bars will bend and break
    My fair lady.
    Build it up with gold and silver
    Gold and silver, gold and.
    """
```

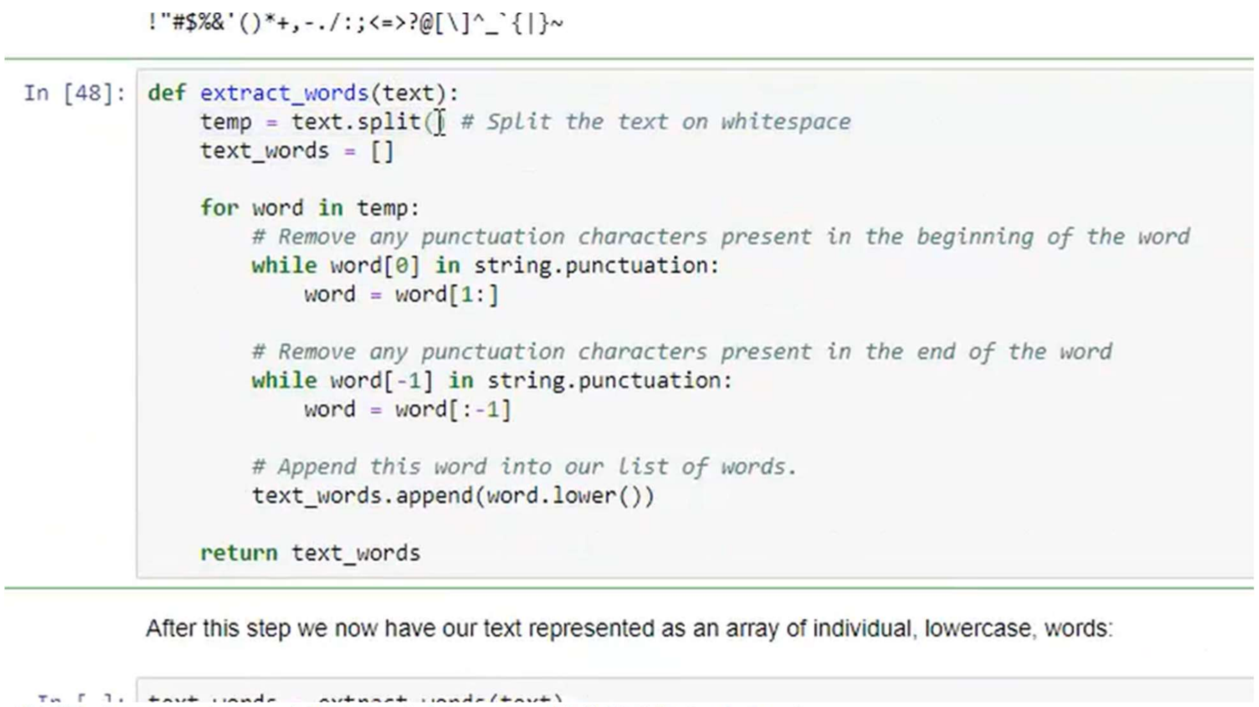
Tokenization

The first step in any analysis is to tokenize the text. What this means is that we will extract all the individual words in the text. Assumptions: Text is properly written and that our words are delimited either by white space or punctuation characters.

```python
print(string.punctuation)
```

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

```python
def extract_words(text):
    temp = text.split() # Split the text on whitespace
    text_words = []

    for word in temp:
        # Remove any punctuation characters present in the beginning of the word
        while word[0] in string.punctuation:
            word = word[1:]

        # Remove any punctuation characters present in the end of the word
        while word[-1] in string.punctuation:
            word = word[:-1]

        # Append this word into our list of words.
        text_words.append(word.lower())
```

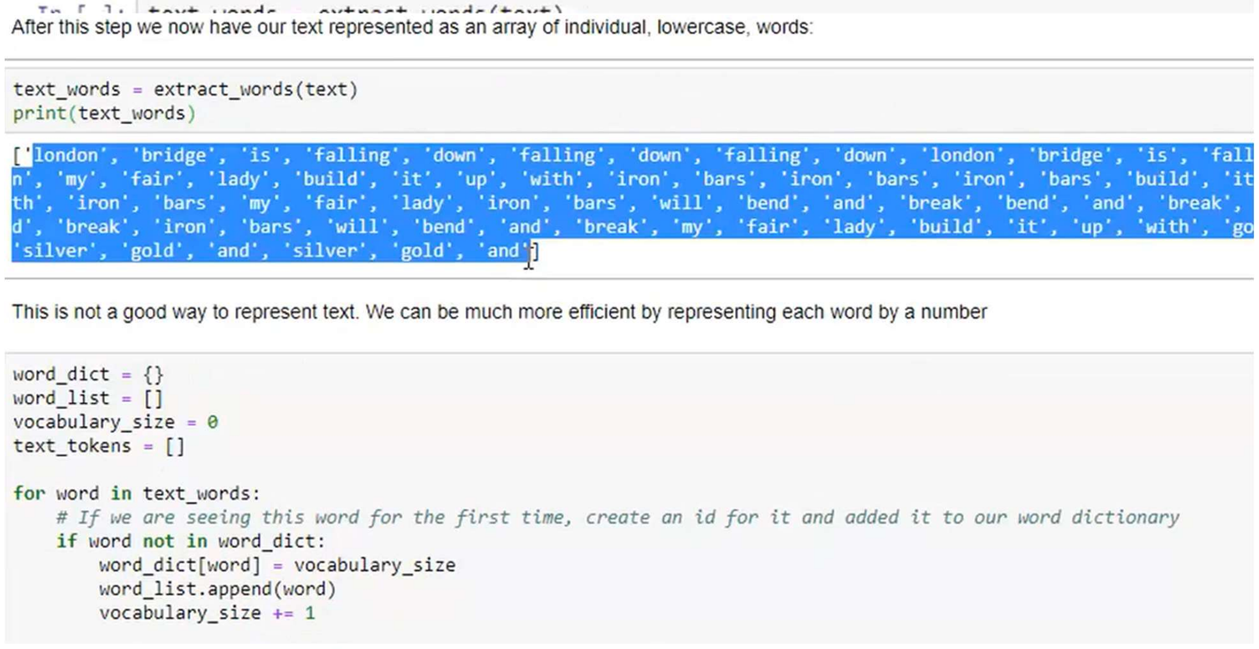


```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

```
In [48]: def extract_words(text):
             temp = text.split() # Split the text on whitespace
             text_words = []

             for word in temp:
                 # Remove any punctuation characters present in the beginning of the word
                 while word[0] in string.punctuation:
                     word = word[1:]

                 # Remove any punctuation characters present in the end of the word
                 while word[-1] in string.punctuation:
                     word = word[:-1]

                 # Append this word into our list of words.
                 text_words.append(word.lower())

             return text_words
```

After this step we now have our text represented as an array of individual, lowercase, words:

```
In [ ]: text_words = extract_words(text)
```



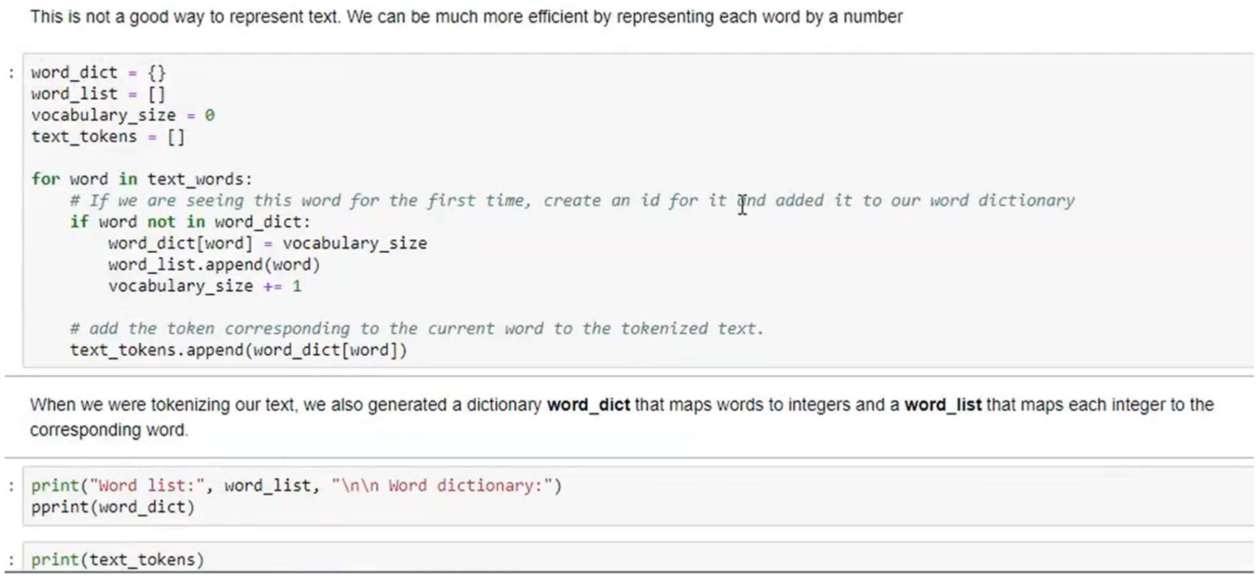
After this step we now have our text represented as an array of individual, lowercase, words:

```
text_words = extract_words(text)
print(text_words)
```

```
['london', 'bridge', 'is', 'falling', 'down', 'falling', 'down', 'falling', 'down', 'london', 'bridge', 'is', 'fall
n', 'my', 'fair', 'lady', 'build', 'it', 'up', 'with', 'iron', 'bars', 'iron', 'bars', 'iron', 'bars', 'build', 'it
th', 'iron', 'bars', 'my', 'fair', 'lady', 'iron', 'bars', 'will', 'bend', 'and', 'break', 'bend', 'and', 'break',
d', 'break', 'iron', 'bars', 'will', 'bend', 'and', 'break', 'my', 'fair', 'lady', 'build', 'it', 'up', 'with', 'go
'silver', 'gold', 'and', 'silver', 'gold', 'and']
```

This is not a good way to represent text. We can be much more efficient by representing each word by a number

```
word_dict = {}
word_list = []
vocabulary_size = 0
text_tokens = []

for word in text_words:
    # If we are seeing this word for the first time, create an id for it and added it to our word dictionary
    if word not in word_dict:
        word_dict[word] = vocabulary_size
        word_list.append(word)
        vocabulary_size += 1
```



This is not a good way to represent text. We can be much more efficient by representing each word by a number

```
: word_dict = {}
  word_list = []
  vocabulary_size = 0
  text_tokens = []

  for word in text_words:
      # If we are seeing this word for the first time, create an id for it and added it to our word dictionary
      if word not in word_dict:
          word_dict[word] = vocabulary_size
          word_list.append(word)
          vocabulary_size += 1

      # add the token corresponding to the current word to the tokenized text.
      text_tokens.append(word_dict[word])
```

When we were tokenizing our text, we also generated a dictionary **word_dict** that maps words to integers and a **word_list** that maps each integer to the corresponding word.

```
: print("Word list:", word_list, "\n\n Word dictionary:")
  pprint(word_dict)
```

```
: print(text_tokens)
```

```
print("Word list:", word_list, "\n\n Word dictionary:")
pprint(word_dict)
```

```
Word list: ['london', 'bridge', 'is', 'falling', 'down', 'my', 'fair', 'lady', 'build', 'it', 'up', 'with', 'iron', 'bars', 'wi
ll', 'bend', 'and', 'break', 'gold', 'silver']

 Word dictionary:
{'and': 16,
 'bars': 13,
 'bend': 15,
 'break': 17,
 'bridge': 1,
 'build': 8,
 'down': 4,
 'fair': 6,
 'falling': 3,
 'gold': 18,
 'iron': 12,
 'is': 2,
 'it': 9,
 'lady': 7,
 'london': 0,
 'my': 5,
 'silver': 19,
 'up': 10,
 'will': 14,
 'with': 11}
```

```
print(text_tokens)
```

```
[0, 1, 2, 3, 4, 3, 4, 3, 4, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 12, 13, 12, 13, 8, 9, 10, 11, 12, 13, 5, 6, 7, 12, 1
3, 14, 15, 16, 17, 15, 16, 17, 15, 16, 17, 12, 13, 14, 15, 16, 17, 5, 6, 7, 8, 9, 10, 11, 18, 16, 19, 18, 16, 19, 18, 16]
```

This representation is convenient for memory reasons it has some severe limitations. Perhaps the most important of which is the fact that computers naturally assume that numbers can be operated on mathematically (by addition, subtraction, etc) in a way that doesn't match our understanding of words.

One-hot encoding

One typical way of overcoming this difficulty is to represent each word by a one-hot encoded vector where every element is zero except the one corresponding to a specific word.

Type *Markdown* and LaTeX: α^2

```
def one_hot(word, word_dict):
    """
        Generate a one-hot encoded vector corresponding to *word*
    """

    vector = np.zeros(len(word_dict))
    vector[word_dict[word]] = 1

    return vector
```

```
    vector = np.zeros(len(word_dict))
    vector[word_dict[word]] = 1

    return vector
```

```
gold_hot = one_hot("gold", word_dict)
print(gold_hot)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```

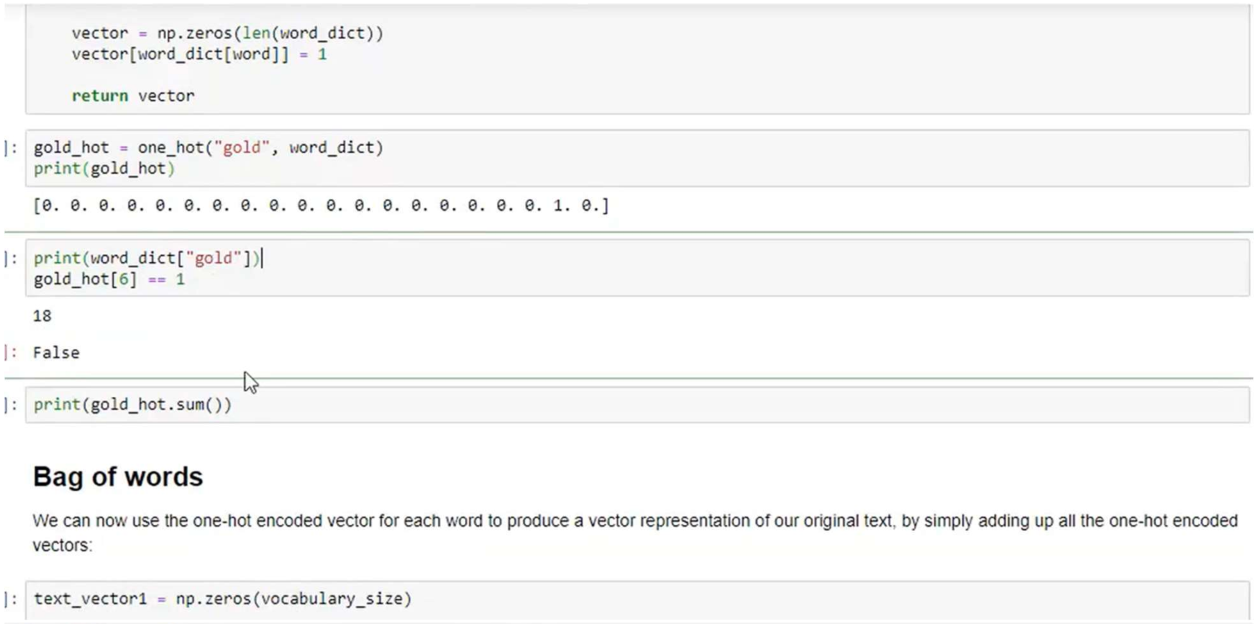
```
print(word_dict["gold"])
gold_hot[6] == 1
```

```
print(gold_hot.sum())
```

Bag of words

We can now use the one-hot encoded vector for each word to produce a vector representation of our original text, by simply adding up all the one-hot encoded vectors:

```
text_vector1 = np.zeros(vocabulary_size)

for word in text_words:
    hot_word = one_hot(word, word_dict)
    text_vector1 += hot_word
```



```
    vector = np.zeros(len(word_dict))
    vector[word_dict[word]] = 1

    return vector
```

]:
```
gold_hot = one_hot("gold", word_dict)
print(gold_hot)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```

]:
```
print(word_dict["gold"])
gold_hot[6] == 1
```

```
18
```

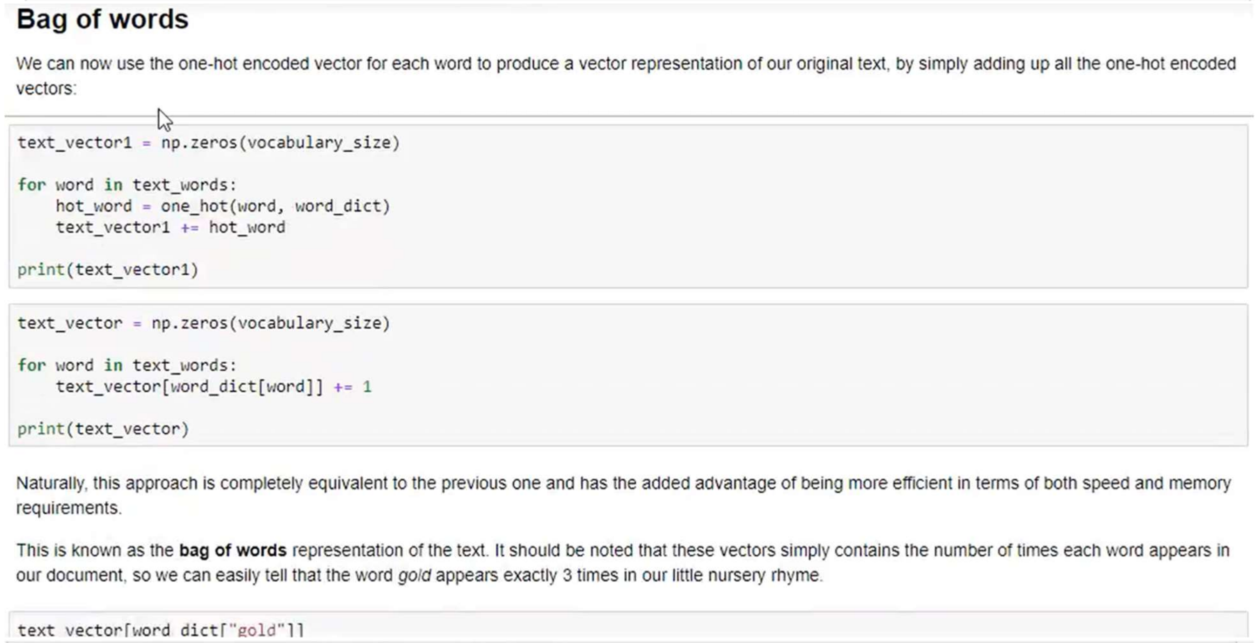
]: False

]:
```
print(gold_hot.sum())
```

Bag of words

We can now use the one-hot encoded vector for each word to produce a vector representation of our original text, by simply adding up all the one-hot encoded vectors:

]:
```
text_vector1 = np.zeros(vocabulary_size)
```



Bag of words

We can now use the one-hot encoded vector for each word to produce a vector representation of our original text, by simply adding up all the one-hot encoded vectors:

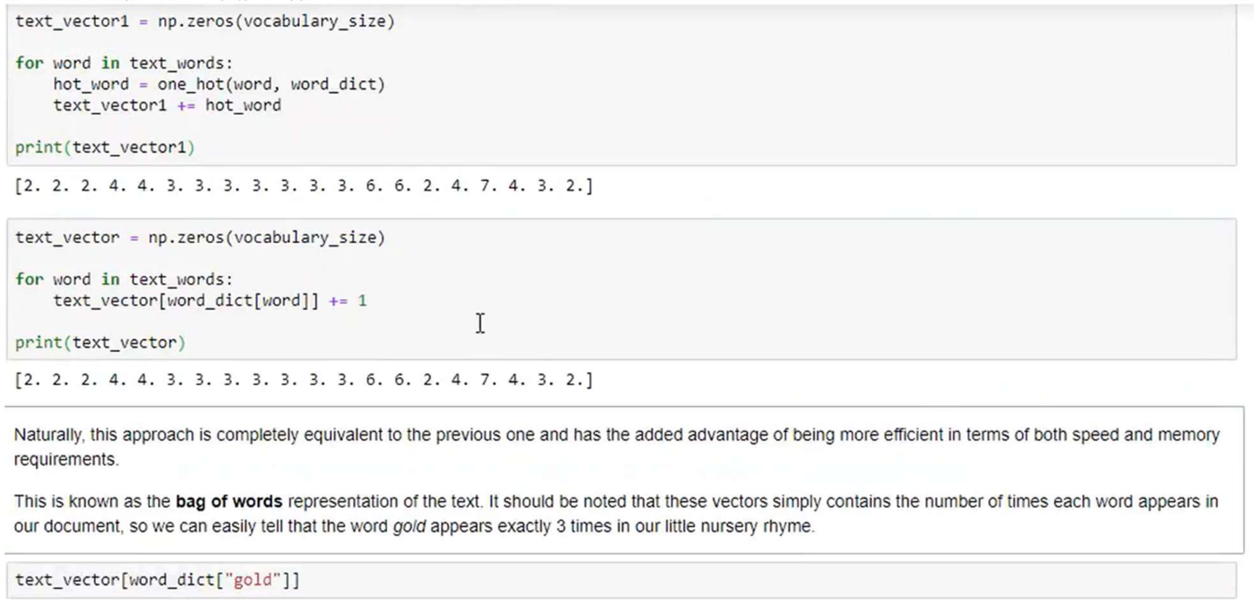
```
text_vector1 = np.zeros(vocabulary_size)

for word in text_words:
    hot_word = one_hot(word, word_dict)
    text_vector1 += hot_word

print(text_vector1)
```

```
text_vector = np.zeros(vocabulary_size)

for word in text_words:
    text_vector[word_dict[word]] += 1

print(text_vector)
```

Naturally, this approach is completely equivalent to the previous one and has the added advantage of being more efficient in terms of both speed and memory requirements.

This is known as the **bag of words** representation of the text. It should be noted that these vectors simply contains the number of times each word appears in our document, so we can easily tell that the word *gold* appears exactly 3 times in our little nursery rhyme.

```
text_vector[word_dict["gold"]]
```



```
text_vector1 = np.zeros(vocabulary_size)

for word in text_words:
    hot_word = one_hot(word, word_dict)
    text_vector1 += hot_word

print(text_vector1)
```

```
[2. 2. 2. 4. 4. 3. 3. 3. 3. 3. 3. 3. 6. 6. 2. 4. 7. 4. 3. 2.]
```

```
text_vector = np.zeros(vocabulary_size)

for word in text_words:
    text_vector[word_dict[word]] += 1

print(text_vector)
```

```
[2. 2. 2. 4. 4. 3. 3. 3. 3. 3. 3. 3. 6. 6. 2. 4. 7. 4. 3. 2.]
```

Naturally, this approach is completely equivalent to the previous one and has the added advantage of being more efficient in terms of both speed and memory requirements.

This is known as the **bag of words** representation of the text. It should be noted that these vectors simply contains the number of times each word appears in our document, so we can easily tell that the word *gold* appears exactly 3 times in our little nursery rhyme.

```
text_vector[word_dict["gold"]]
```

```
text_words
```

```
word_counts = Counter(text_words)
pprint(word_counts)
```

From which we can easily generate the **text_vector** and **word_dict** data structures:

```
items = list(word_counts.items())

# Extract word dictionary and vector representation
word_dict2 = dict([[items[i][0], i] for i in range(len(items))])
text_vector2 = [items[i][1] for i in range(len(items))]
```

```
word_counts['gold']
```

```
text_vector
```

```
text_vector
```

```
array([2., 2., 2., 4., 4., 3., 3., 3., 3., 3., 3., 3., 6., 6., 2., 4., 7.,
       4., 3., 2.])
```

```
print("Text vector:", text_vector2, "\n\nWord dictionary:")
pprint(word_dict2)
```

The results using this approach are slightly different than the previous ones, because the words are mapped to different integer ids but the corresponding values are the same:

```
for word in word_dict.keys():
    if text_vector[word_dict[word]] != text_vector2[word_dict2[word]]:
        print("Error!")
```

As expected, there are no differences!

Term Frequency

The bag of words vector representation introduced above relies simply on the frequency of occurence of each word. Following a long tradition of giving fancy names to simple ideas, this is known as **Term Frequency**.

ong tradition of giving fancy names to simple ideas, this is known as **Term Frequency**. Intuitively, we expect the the frequency with which a given word is mentioned should correspond to the relevance of that word for the piece of text we are considering. For example, Iron, bars, and are pretty important word in our little nursery rhyme and indeed it is the one that occurs the most often:

```
sorted(items, key=lambda x:x[1], reverse=True)
```

However, it's hard to draw conclusions from such a small piece of text. Let us consider a significantly larger piece of text, the first 100 MB of the english Wikipedia from: http://mattmahoney.net/dc/textdata. For the sake of convenience, text8.gz has been included in this repository in the **data/** directory. We start by loading it's contents into memory as an array of words:

```
data = []

for line in gzip.open("data/text8.gz", 'rt'):
    data.extend(line.strip().split())
```

And the top 10 most common words

```python
counts = Counter(data)

sorted_counts = sorted(list(counts.items()), key=lambda x: x[1], reverse=True)

for word, count in sorted_counts[:10]:
    print(word, count)
```

Surprisingly, we find that the most common words are not particularly meaningful. Indeed, this is a common occurence in Natural Language Processing. The most frequent words are typically auxiliaries required due to gramatical rules.

On the other hand, there is also a large number of words that occur very infrequently as can be easily seen by glancing at the word freqency distribution.

```python
dist = Counter(counts.values())
dist = list(dist.items())
dist.sort(key=lambda x:x[0])
dist = np.array(dist)

norm = np.dot(dist.T[0], dist.T[1])

plt.loglog(dist.T[0], dist.T[1]/norm)
plt.xlabel("count")
plt.ylabel("P(count)")
plt.title("Word frequency distribution")
```

Surprisingly, we find that the most common words are not particularly meaningful. Indeed, this is a common occurence in Natural Language Processing. The most frequent words are typically auxiliaries required due to gramatical rules.

On the other hand, there is also a large number of words that occur very infrequently as can be easily seen by glancing at the word freqency distribution.

```python
dist = Counter(counts.values())
dist = list(dist.items())
dist.sort(key=lambda x:x[0])
dist = np.array(dist)

norm = np.dot(dist.T[0], dist.T[1])

plt.loglog(dist.T[0], dist.T[1]/norm)
plt.xlabel("count")
plt.ylabel("P(count)")
plt.title("Word frequency distribution")
plt.gcf().set_size_inches(11, 8)
```

Stopwords

```python
stopwords = set([word for word, count in sorted_counts[:100]])

clean_data = []

for word in data:
    if word not in stopwords:
        clean_data.append(word)
```

Stopwords

```
stopwords = set([word for word, count in sorted_counts[:100]])

clean_data = []

for word in data:
    if word not in stopwords:
        clean_data.append(word)

print("Original size:", len(data))
print("Clean size:", len(clean_data))
print("Reduction:", 1-len(clean_data)/len(data))
```

```
clean_data[:50]
```

Term Frequency/Inverse Document Frequency

```
text = """Mary had a little lamb, little lamb,
    little lamb. Mary had a little lamb
    whose fleece was white as snow.
    And everywhere that Mary went
    Mary went, Mary went. Everywhere
```

```
print(text)
```

```
Mary had a little lamb, little lamb,
    little lamb. Mary had a little lamb
    whose fleece was white as snow.
    And everywhere that Mary went
    Mary went, Mary went. Everywhere
    that Mary went,
    The lamb was sure to go
```

```
corpus_text = text.split('.')
corpus_words = []

for document in corpus_text:
    doc_words = extract_words(document)
    corpus_words.append(doc_words)
```

Now our corpus is represented as a list of word lists, where each list is just the word representation of the corresponding sentence

```
print(len(corpus_words))
```

```
pprint(corpus_words)
```

Let us now calculate the number of documents in which each word appears:

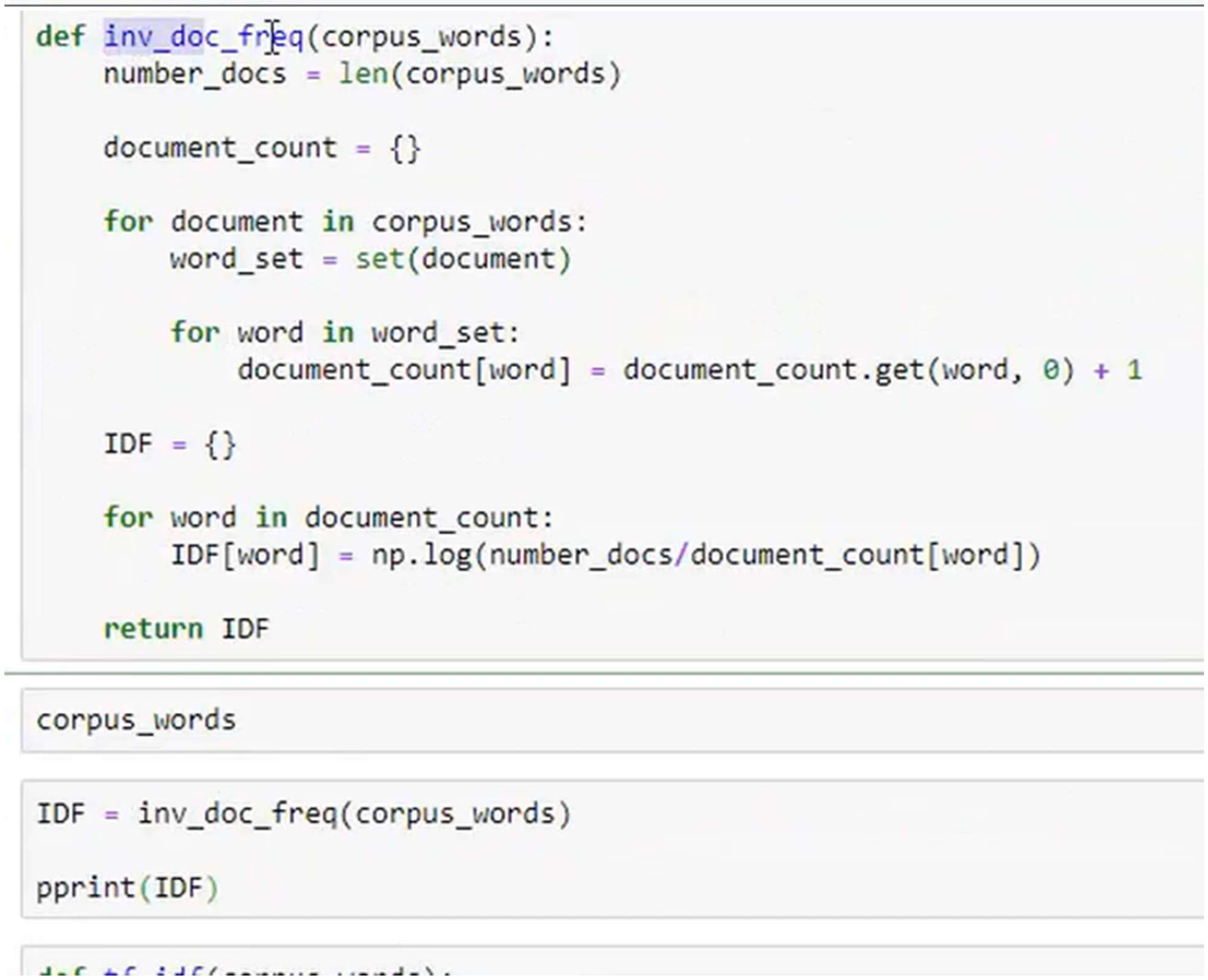
```
document_count = {}
```

Let us now calculate the number of documents in which each word appears:

```
document_count = {}

for document in corpus_words:
    word_set = set(document)

    for word in word_set:
        document_count[word] = document_count.get(word, 0) + 1

pprint(document_count)
```

```
As we can see, the word Mary appears in all 4 of our documents, making it useless when it comes to distinguish between the
different sentences. On the other hand, words like white which appear in only one document are very discriminative. Using this
approach we can define a new quantity, the _Inverse Document Frequency that tells us how frequent a word is across the
documents in a specific corpus
```

```
def inv_doc_freq(corpus_words):
    number_docs = len(corpus_words)

    document_count = {}
```



```
def inv_doc_freq(corpus_words):
    number_docs = len(corpus_words)

    document_count = {}

    for document in corpus_words:
        word_set = set(document)

        for word in word_set:
            document_count[word] = document_count.get(word, 0) + 1

    IDF = {}

    for word in document_count:
        IDF[word] = np.log(number_docs/document_count[word])

    return IDF
```

```
corpus_words
```

```
IDF = inv_doc_freq(corpus_words)

pprint(IDF)
```

```
def tf_idf(corpus_words):
```

```
def tf_idf(corpus_words):
    IDF = inv_doc_freq(corpus_words)

    TFIDF = []

    for document in corpus_words:
        TFIDF.append(Counter(document))

    for document in TFIDF:
        for word in document:
            document[word] = document[word]*IDF[word]

    return TFIDF
```

```
tf_idf(corpus_words)
```

Now we finally have a vector representation of each of our documents that takes the informational contributions of each word into account. Each of these vectors provides us with a unique representation of each document, in the context (corpus) in which it occurs, making it posssible to define the similarity of two documents, etc.