

Some common terms to remember:

- 1. Corpus - Paragrapgh
- 2. Vocabulary - Unique word
- 3. Document - one row
- 4. Word - one word

Bag of words

```
In [1]: import numpy as np
import pandas as pd
```

```
In [2]: df = pd.DataFrame({"text":["people watch dswithbappy",
                                "dswithbappy watch dswithbappy",
                                "people write comment",
                                "dswithbappy write comment"],"output":[1,1,0,0]})

df
```

Out[2]:

	text	output
0	people watch dswithbappy	1
1	dswithbappy watch dswithbappy	1
2	people write comment	0
3	dswithbappy write comment	0

```
In [3]: from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
```

```
In [4]: bow = cv.fit_transform(df['text'])
```

```
In [5]: #vocabulary
print(cv.vocabulary_)

{'people': 2, 'watch': 3, 'dswithbappy': 1, 'write': 4, 'comment': 0}
```

```
In [10]: bow.toarray()
```

```
Out[10]: array([[0, 1, 1, 1, 0],
                [0, 2, 0, 1, 0],
                [1, 0, 1, 0, 1],
                [1, 1, 0, 0, 1]])
```

```
In [ ]: print(bow[0].toarray())
print(bow[1].toarray())
print(bow[2].toarray())

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

```
In [8]: # new
cv.transform(['Bappy watch dswithbappy']).toarray()
```

```
Out[8]: array([[0, 1, 0, 1, 0]])
```

```
In [ ]: X = bow.toarray()
y = df['output']
```

```
In [ ]:
```

N-grams

```
In [11]: df = pd.DataFrame({"text":["people watch dswithbappy",
                                "dswithbappy watch dswithbappy",
                                "people write comment",
                                "dswithbappy write comment"],"output":[1,1,0,0]})

df
```

Out[11]:

	text	output
0	people watch dswithbappy	1
1	dswithbappy watch dswithbappy	1
2	people write comment	0
3	dswithbappy write comment	0

```
In [12]: # BI grams
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(ngram_range=(2,2))
```

```
In [13]: bow = cv.fit_transform(df['text'])
```

```
In [14]: print(cv.vocabulary_)

{'people watch': 2, 'watch dswithbappy': 4, 'dswithbappy watch': 0, 'people write': 3, 'write comment': 5, 'dswithbappy write': 1}
```

```
In [ ]: print(bow[0].toarray())
print(bow[1].toarray())
print(bow[2].toarray())

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

```
In [15]: #Ti gram
# BI grams
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(ngram_range=(3,3))
```

```
In [16]: bow = cv.fit_transform(df['text'])
```

```
In [ ]: print(cv.vocabulary_)

{'people watch dswithbappy': 2, 'dswithbappy watch dswithbappy': 0, 'people write comme
nt': 3, 'dswithbappy write comment': 1}
```

```
In [ ]: print(bow[0].toarray())
print(bow[1].toarray())
print(bow[2].toarray())

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

TF-IDF (Term frequency- Inverse document frequency)

```
In [ ]: df = pd.DataFrame({"text":["people watch dswithbappy",
                                "dswithbappy watch dswithbappy",
                                "people write comment",
                                "dswithbappy write comment"],"output":[1,1,0,0]})

df
```

Out[20]:

	text	output
0	people watch dswithbappy	1
1	dswithbappy watch dswithbappy	1
2	people write comment	0
3	dswithbappy write comment	0

```
In [ ]: from sklearn.feature_extraction.text import TfidfVectorizer
tfidf= TfidfVectorizer()
```

```
In [ ]: tfidf.fit_transform(df['text']).toarray()
```

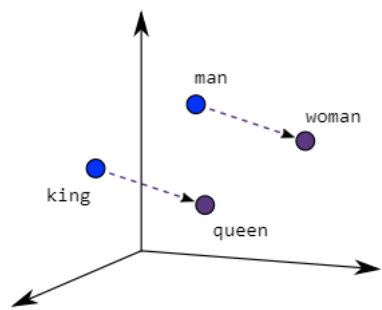
Out[22]: array([[0. , 0.49681612, 0.61366674, 0.61366674, 0.],
 [0. , 0.8508161 , 0. , 0.52546357, 0.],
 [0.57735027, 0. , 0.57735027, 0. , 0.57735027],
 [0.61366674, 0.49681612, 0. , 0. , 0.61366674]])

```
In [ ]: print(tfidf.idf_)

[1.51082562 1.22314355 1.51082562 1.51082562 1.51082562]
```

Word2Vec

Word2Vec creates vectors of the words that are distributed numerical representations of word features – these word features could comprise of words that represent the context of the individual words present in our vocabulary. Word embeddings eventually help in establishing the association of a word with another similar meaning word through the created vector.



```
In [1]: import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

```
In [2]: import numpy as np
import pandas as pd
import gensim
import os
```

```
In [3]: !pip install --upgrade gensim --user

Requirement already satisfied: gensim in c:\users\shehryar gondal\appdata\roaming\python\python310\site-packages (4.3.2)
Requirement already satisfied: smart-open>=1.8.1 in d:\anaconda setup\lib\site-packages (from gensim) (5.2.1)
Requirement already satisfied: numpy>=1.18.5 in d:\anaconda setup\lib\site-packages (from gensim) (1.23.5)
Requirement already satisfied: scipy>=1.7.0 in d:\anaconda setup\lib\site-packages (from gensim) (1.10.0)

WARNING: Ignoring invalid distribution -orch (d:\anaconda setup\lib\site-packages)
WARNING: Ignoring invalid distribution -orch (d:\anaconda setup\lib\site-packages)
WARNING: Ignoring invalid distribution -orch (d:\anaconda setup\lib\site-packages)
WARNING: Ignoring invalid distribution -orch (d:\anaconda setup\lib\site-packages)
WARNING: Ignoring invalid distribution -orch (d:\anaconda setup\lib\site-packages)
```

```
In [4]: from nltk import sent_tokenize
from gensim.utils import simple_preprocess
import nltk
nltk.download('punkt')

[nltk_data] Downloading package punkt to C:\Users\Shehryar
[nltk_data]   Gondal\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping tokenizers\punkt.zip.
```

Out[4]: True

Building Blocks of Word Embedding

Dimensionality Reduction

Large word vectors are

Understanding Word Embedding: Basics and Significance

What is Word Embedding?

Word Embedding is a technique that represents words in a continuous, multi-dimensional vector space where words that have similar meaning are closer to each other.

Why is it significant?

Word Embedding helps to better capture the meaning of natural language expressions and it helps to solve many of the challenges that are encountered when processing text data.

How does it work?

Word Embedding algorithms create vectors for each word in a corpus of text. Each dimension in the vector represents a different feature of the word, such as its part of speech, its semantic meaning, or its context in the text.

What are the benefits?

Word Embedding can improve the accuracy of NLP models, enhance the expressiveness of language, and enable machines to understand natural language more effectively.

```
In [ ]:
```

```
In [6]: import os
from nltk.tokenize import sent_tokenize
from gensim.utils import simple_preprocess # Assuming you're using gensim for preprocessing

# Initialize an empty list to store the tokenized sentences
story = []

# Specify the directory containing your text files
```

Understanding Word Embedding: Basics and Significance

What is Word Embedding?

Word Embedding is a technique that represents words in a continuous, multi-dimensional vector space where words that have similar meaning are closer to each other.

Why is it significant?

Word Embedding helps to better capture the meaning of natural language expressions and it helps to solve many of the challenges that are encountered when processing text data.

How does it work?

Word Embedding algorithms create vectors for each word in a corpus of text. Each dimension in the vector represents a different feature of the word, such as its part of speech, its semantic meaning, or its context in the text.

What are the benefits?

Word Embedding can improve the accuracy of NLP models, enhance the expressiveness of language, and enable machines to understand natural language more effectively.

```
import os
from nltk.tokenize import sent_tokenize
from gensim.utils import simple_preprocess # Assuming you're using gensim for preprocessing

# Initialize an empty list to store the tokenized sentences
story = []

# Specify the directory containing your text files
data_directory = 'data1/'

# Iterate through files in the directory
for filename in os.listdir(data_directory):
    if filename == '.ipynb_checkpoints':
        continue # Skip the .ipynb_checkpoints directory if present
    with open(os.path.join(data_directory, filename), 'r', encoding='utf-8') as file:
        corpus = file.read()
        raw_sentences = sent_tokenize(corpus)
        for sentence in raw_sentences:
            story.append(simple_preprocess(sentence))
```

story

```
[[ 'game',
  'of',
  'thrones',
  'book',
  'one',
  'of',
  'song',
  'of',
  'ice',
  'and',
  'fire',
  'by',
  'george',
  'martin',
  'prologue',
  'we',
  'should',
  'start',
  'back',
  ,,
```

```
len(story)
```

8356

```
story
story[0]
[[['game',
  ['game',
    ['of',
      'thrones',
      'books',
      'book',
      'of',
      'of',
      'of',
      'song',
      'song',
      'of',
      'ice',
      'ice',
      'and',
      'and',
      'fire',
      'fire',
      'by',
      'by',
      'george',
      'martin',
      'martin',
      'prologue',
      'prologue',
      'we',
      'we',
      'should',
      'should',
      'start',
      'start',
      'back',
      'back',
      'back',
      'gared',
      'gared',
      'urged',
      'as',
      'the',
      'woods',
      'began',
      'to',
      'grow',
      'dark',
      'around',
      'them']]]]
```

```
model = gensim.models.Word2Vec(  
    window=10,  
    min_count=2  
)
```

```
In [9]: story
In [10]: story[0]
Out[9]: ['game',
Out[10]: ['game',
        'game',
        'throne',
        'books',
        'books',
        'books',
        'of',
        'song',
        'songs',
        'of',
        'ice',
        'land',
        'apple',
        'fire',
        'by',
        'george',
        'george',
        'martin',
        'martin',
        'prologue',
        'we',
        'we should',
        'should',
        'start',
        'back',
        'back',
        'gared',
        'urged',
        'as',
        'the',
        'woods',
        'began',
        'to',
        'grow',
        'dark',
        'around',
        'them']

In [11]: model = gensim.models.Word2Vec(
        window=10,
        min_count=2
    )

In [12]: model.build_vocab(story)

In [13]: model.train(story, total_examples=model.corpus_count, epochs=model.epochs)

Out[13]: (322595, 447775)

In [14]: model.wv.most_similar('daenerys')

Out[14]: [('still', 0.9993206262588501),
          ('two', 0.9992164373397827),
          ('dothraki', 0.9992116689682007),
          ('an', 0.9992004036903381),
          ('above', 0.9991950988769531),
          ('while', 0.9991921186447144),
          ('other', 0.9991881251335144),
          ('castle', 0.999164879322052),
          ('then', 0.9991580247879028),
          ('three', 0.9991518259048462)]

In [15]: model.wv.similarity('arya', 'sansa')

Out[15]: 0.9997436

In [16]: model.wv['deep'].shape

Out[16]: (100,)

In [17]: vec = model.wv.get_normed_vectors()

In [18]: vec

Out[18]: array([[ -0.12478705,  0.06850963,  0.10970236, ..., -0.09639692,
                  0.07652823,  0.0394804 ],
                [ -0.11689644,  0.06340289,  0.10848914, ..., -0.09198184,
                  0.07238222,  0.03124547],
                [ -0.07155877,  0.03460521,  0.09148999, ..., -0.08466641,
                  0.06697579,  0.00204907],
                ...,
                ['tyrion', 0.05747115,  0.14610448,  0.13229322, ..., -0.10777298,
                  0.06255373,  0.06095371],
                'hand', [-0.07182547,  0.06408214,  0.06909133, ..., -0.10851807,
                  0.05367512, -0.00807478],
                'we', [ 0.00190406,  0.09480352,  0.01514295, ..., -0.07081555,
                  0.07665265, -0.00382668]], dtype=float32)

In [19]: model.wv.get_normed_vectors().shape
Out[19]: (3040, 100)

In [20]: y = model.wv.index_to_key
Out[20]: 'by',
        'see',
        'told',
        'boy',
        'father',
        'lenny',
        'only',
        'sansa',
        'brother',
        'an',

In [21]: len(y)
Out[21]: 3840

In [23]: from sklearn.decomposition import PCA

In [24]: pca = PCA(n_components=3)

In [25]: X = pca.fit_transform(model.wv.get_normed_vectors())

In [26]: X
Out[26]: array([[ -1.4590222e-02, -2.1133895e-01, -4.7438904e-03],
                [ -2.7601445e-02, -1.6377506e-01, -9.5280139e-03],
                [ -5.2432995e-02,  2.0638121e-02,  1.0925694e-04],
                ...,
                ...])
```

```
In [22]: y = [[0.06697579, -0.00204907],
...,
'tyrior', 0.05747115, 0.14610448, 0.13229322, ..., -0.10777298,
'hand', 0.06255373, 0.06095371],
'we', [-0.07182547, 0.06408214, 0.06909133, ..., -0.10851807,
'then', 0.05367512, -0.00807478],
'do', [ 0.00190406, 0.09480352, 0.01514295, ..., -0.07081555,
'than', 0.07665265, -0.00382668]], dtype=float32)
'down',
'never',
'page', 100)
'by',
'see',
'told',
y, model.wv.index_to_key
'boy',
'father',
'len(y)',
'only',
'sansa',
'brother',
'an',

In [19]: model.wv.get_normed_vectors().shape
Out[19]: (3840, 100)

In [20]:
y, model.wv.index_to_key
'boy',
'father',
'len(y)',
'only',
'sansa',
'brother',
'an',

In [21]: len(y)
Out[21]: 3840

In [23]:
from sklearn.decomposition import PCA

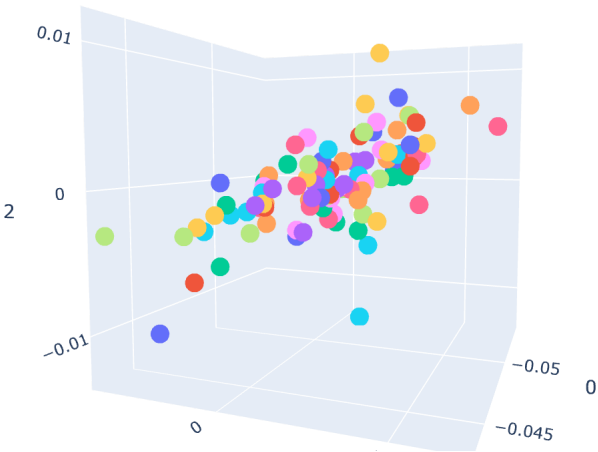
In [24]:
pca = PCA(n_components=3)

In [25]:
X = pca.fit_transform(model.wv.get_normed_vectors())

In [26]: X
Out[26]: array([[ -1.4590222e-02, -2.1133895e-01, -4.7438904e-03],
[ -2.7601445e-02, -1.6377506e-01, -9.5280139e-03],
[ -5.2432995e-02,  2.0638121e-02,  1.0925694e-04],
...,
[  1.5823811e-01,  2.3719370e-02,  1.5382897e-02],
[ -2.7450187e-02, -6.2797256e-02, -1.9610928e-02],
[  3.5188053e-02,  1.1956904e-01, -2.4944900e-03]], dtype=float32)

In [27]: X.shape
Out[27]: (3840, 3)

In [28]: import plotly.express as px
fig = px.scatter_3d(X[200:300],x=0,y=1,z=2, color=y[200:300])
fig.show()
```



```
In [ ]:
```