DATA SOCIETY®

Advanced text mining - part 3

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Welcome back!

- While you wait for class to get started, draft a “tweet” of less than 280 characters that summarizes what you learned in the last session
- Share it in the chat box (and on Twitter too, if you like)

Module completion checklist

Objective	Complete
Summarize feature engineering and word embeddings	
Create a Gensim Word2vec model to view similar words	
Download and load GloVe word embeddings	
Use pre-trained word embeddings to generate text features for model development	
Recap cosine similarity and how it applies to our document embeddings matrix	
Compute cosine similarity and find similar documents	

Import packages

- Let's import the packages

```
# Helper packages.
import os
import pickle
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
import nltk
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

```
# Packages for loading pre-trained word-embedding model
import gensim
from gensim.models import Word2Vec
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `booz-allen-hamilton` folder

```
from pathlib import Path
# Set `home_dir` to the root directory of your computer.
home_dir = Path.home()

# Set `main_dir` to the location of your `booz-allen-hamilton` folder.
main_dir = home_dir / "Desktop" / "booz-allen-hamilton"

# Make `data_dir` from the `main_dir` and remainder of the path to data directory.
data_dir = main_dir / "data"
```

Working directory

- Set working directory to the `data_dir` variable we set

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/Desktop/booz-allen-hamilton/data
```

Loading text data

- The corpus in the pretrained models used later in this course is **not stemmed**
- To remain consistent, we need to process all the cleaning steps (as performed in our earlier module) but this time without stemming the corpus

```
# Load corpus from a csv (for Mac).
NYT = pd.read_csv('NYT_article_data.csv')
```

Recap: corpus pre-processing steps

- Let's load the snippets into a variable and tokenize each snippet

```
num_docs = len(NYT["snippet"])
print(num_docs)

# Tokenize each text into a large list of tokenized snippets.
```

```
250
```

```
NYT_tokenized = [word_tokenize(snippet) for snippet in NYT["snippet"]]
```

Cleaning function

- `prep_text()` performs the following tasks:
  - converts tokens to lower case, removes stop words, punctuation, and non-alphabetical characters
  - converts clean tokens back to snippets

```
def prep_text(text_tokenized):
  # Process words in all snippets.
  clean_text = [None]*len(text_tokenized)

  for i in range(len(text_tokenized)):
    # 1. Convert to lower case.
    text = [word.lower() for word in text_tokenized[i]]
    # 2. Remove stop words.
    text = [word for word in text if not word in stop_words]
    # 3. Remove punctuation and any non-alphabetical characters.
    text = [word for word in text if word.isalpha()]
    clean_text[i] = text

  clean_text_list = [' '.join(snippet) for snippet in clean_text]

  return clean_text_list, clean_text
```

Prep NYT text for analysis

- Let's create 2 objects:
  - `NYT_clean` - a list of clean tokens for each snippet
  - `NYT_clean_list` - a list of clean snippets

```
NYT_clean_list, NYT_clean = prep_text(NYT_tokenized)
print(NYT_clean[:3])
```

```
[['pakistan', 'struggling', 'batsmen', 'must', 'find', 'way', 'handle', 'south', 'africa', 'potent',
'pace', 'attack', 'claw', 'way', 'back', 'series', 'second', 'test', 'starts', 'likely', 'lively',
'newlands', 'wicket', 'thursday'], ['national', 'football', 'league', 'microscope', 'lack', 'minority',
'head', 'coaches', 'recent', 'slew', 'firings', 'league'], ['hitting', 'hot', 'streak', 'right',
'time', 'goal', 'golf', 'top', 'male', 'professionals', 'year', 'new', 'calendar', 'crams', 'major',
'championships', 'super', 'busy', 'stretch']]
```

```
print(NYT_clean_list[:3])
```

```
['pakistan struggling batsmen must find way handle south africa potent pace attack claw way back series
second test starts likely lively newlands wicket thursday', 'national football league microscope lack
minority head coaches recent slew firings league', 'hitting hot streak right time goal golf top male
professionals year new calendar crams major championships super busy stretch']
```

Recap: create a DTM

```
# Initialize `CountVectorizer`.
vec = CountVectorizer()
```

```
# Transform the list of snippets into DTM.
X = vec.fit_transform(NYT_clean_list)
print(X.toarray()) #<- show output as a matrix
```

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

- To get a list of names of columns (i.e. the *unique terms* in our corpus), we can use a utility method `.get_feature_names()`

```
print(vec.get_feature_names()[:10])
```

```
['abducted', 'able', 'abo', 'absentee', 'abuse', 'abusing', 'academic', 'accepted', 'access',
'accessories']
```

Recap: create a DTM (cont'd)

- Let's convert the matrix into a dataframe, where rows are IDs of the snippets and columns are unique words that appear in those snippets

```
# Convert the matrix into a pandas dataframe for easier manipulation.
DTM_not_stemmed = pd.DataFrame(X.toarray(), columns = vec.get_feature_names())
print(DTM_not_stemmed.head())
```

```
   abducted  able  abo  absentee  abuse  ...  york  young  yuan  zimbabwe  zykera
0         0     0    0         0      0  ...     0      0     0         0       0
1         0     0    0         0      0  ...     0      0     0         0       0
2         0     0    0         0      0  ...     0      0     0         0       0
3         0     0    0         0      0  ...     0      0     0         0       0
4         0     0    0         0      0  ...     0      0     0         0       0

[5 rows x 2272 columns]
```

Feature engineering

- The steps above were a preparation for our next big task: **feature engineering**
- It is the process of **creating new meaningful features from existing raw ones that help facilitate the machine learning algorithm**
- It seeks the **best representation of the data** and is focused on **transforming data while capturing its inherent structure**
- If done correctly, the transformed inputs to the machine learning algorithm help to improve overall model performance and reduce training time

Item ID	Number of items	Unit price	Total price
IA_1	2	$34	$68
IA_2	43	$52	$2,236
IA_3	61	$21	$1,281
IA_4	12	$19	$228

Feature engineering in text

- Feature engineering for unstructured, textual data is what powers the entire field of text analysis and NLP
  - In order to analyse text using ML and DL techniques, we need to convert raw text into numeric representations which can be understood by algorithms
  - In a way, *you have already been doing feature engineering all along* by creating DTM/TDM with and without Tf-Idf weights
- One of the most popular methods of extracting meaningful features from text data is through **word embeddings**

Word embeddings in Python

- Word embedding is a language feature engineering technique used for **converting words to numeric vectors**
- These vectors can contain anywhere from tens to hundreds of entries that we call **dimensions**
  - If n is a number of entries/dimensions in such a vector, for example $50 < n < 500$, then a word embedding would look like this

$$W(" \ cat \ ") = (0.2, -0.4, 0.7, \ldots, n)$$

Word embeddings methods

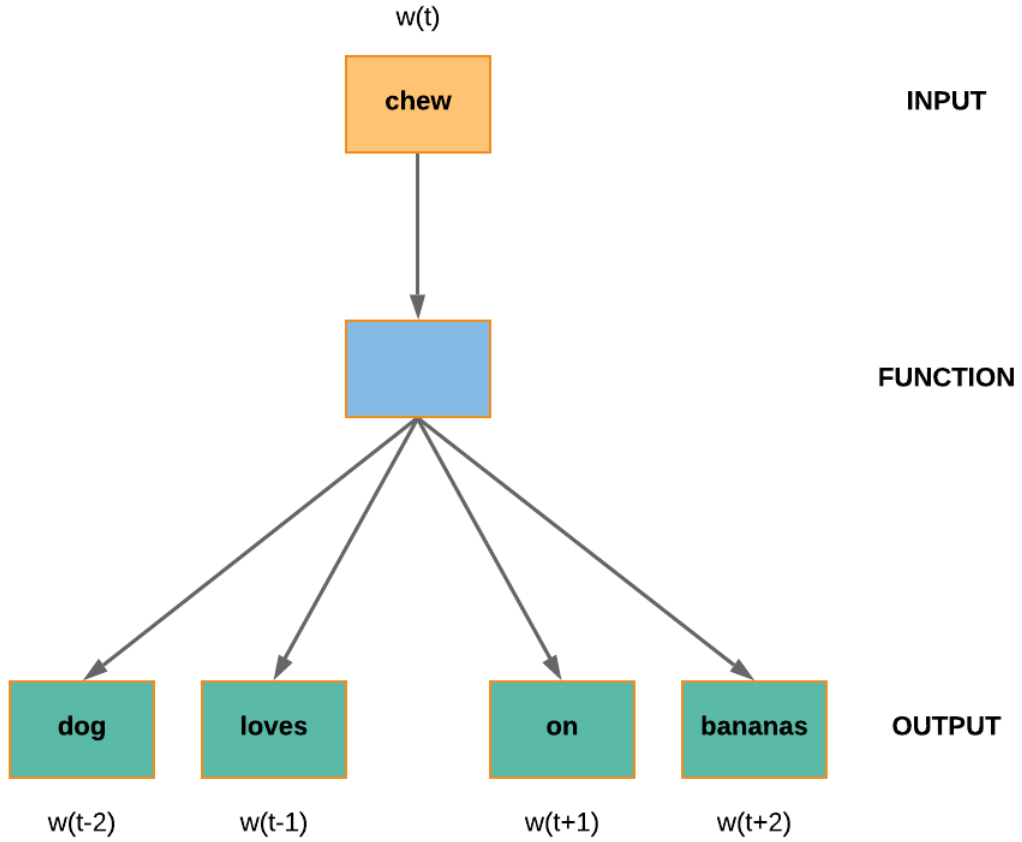
- To use word embeddings, you have two options:
  - Train a custom model using your own data and one of existing word embedding algorithms
  - Use one of the available pre-trained models of your choice for some language - a more popular and robust way to leverage accumulated knowlegde/algorithm base called **transfer learning**

Word embedding methods: Word2Vec

- Word2Vec approach uses a multi-layered neural network algorithm to convert words into vector representations
  - This way, the words that are similar in meaning are close to each other in n-dimensional space, where n refers to the dimensions of the vector
- There are two model types in Word2Vec
  - **Skip-gram** model
  - **Continuous Bag Of Words (CBOW)** model

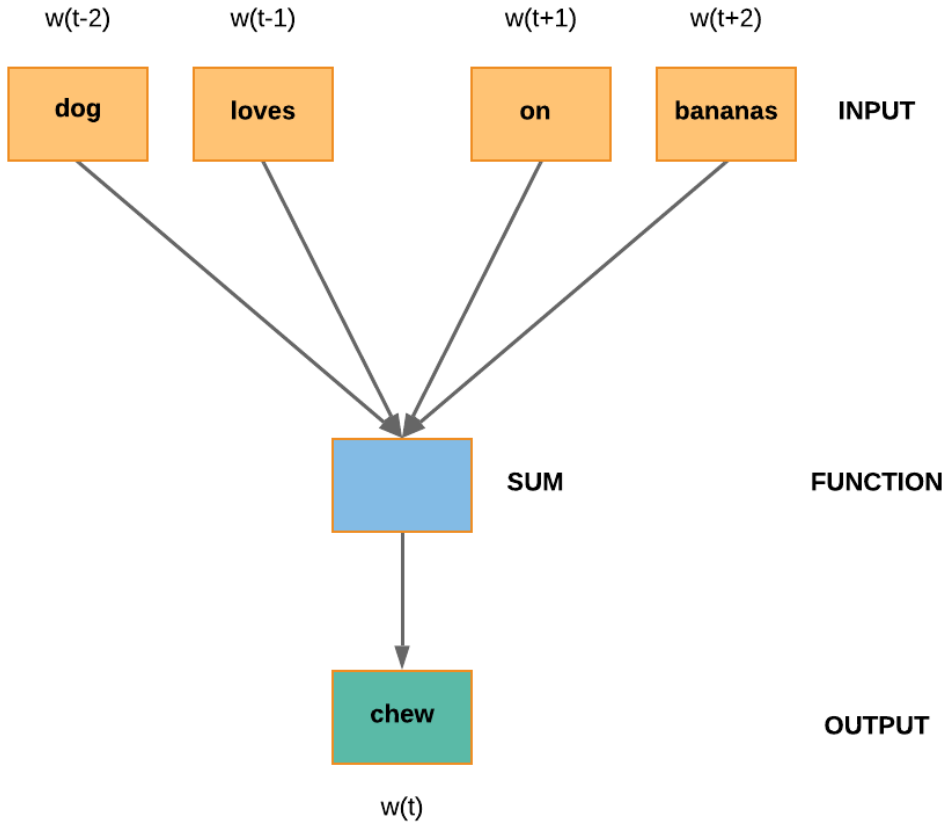
Word2Vec model types: skip-gram

- **Skip-gram** model uses a single word to predict multiple context words
  - For instance, given a sentence *My dog loves to chew on bananas*, the skip-gram model will predict *dog*, *bananas*, and so on given the word *chew* as input



Word2Vec model types: CBOW

- **Continuous Bag Of Words (CBOW)** model takes multiple context words to predict a single word
  - Given a sentence *My dog loves to chew on bananas*, the CBOW model will predict *chew*, given the words *dog*, *bananas*, *loves*, etc. as input
- In this class, we will demonstrate how to train a Word2Vec model using a **Skip-gram** model



Knowledge check 1

Module completion checklist

Objective	Complete
Summarize feature engineering and word embeddings	✔
Create a Gensim Word2vec model to view similar words	
Download and load GloVe word embeddings	
Use pre-trained word embeddings to generate text features for model development	
Recap cosine similarity and how it applies to our document embeddings matrix	
Compute cosine similarity and find similar documents	

gensim.models.Word2Vec

- The `gensim` package we have been using in previous modules also provides access to `Word2Vec` and other word embedding algorithms for training
- We will use the pretrained `Word2vec` to find similar words in NYT snippets

`models.word2vec`

Produce word vectors with deep learning via word2vec's "skip-gram and CBOW models", using either hierarchical softmax or negative sampling [1] [2].

NOTE: There are more ways to get word vectors in Gensim than just Word2Vec. See wrappers for FastText, VarEmbed and WordRank.

The training algorithms were originally ported from the C package https://code.google.com/p/word2vec/ and extended with additional functionality.

For a blog tutorial on gensim word2vec, with an interactive web app trained on GoogleNews, visit http://radimrehurek.com/2014/02/word2vec-tutorial/

Make sure you have a C compiler before installing gensim, to use optimized (compiled) word2vec training (70x speedup compared to plain NumPy implementation [3]).

Initialize a model with e.g.:

```
>>> model = Word2Vec(sentences, size=100, window=5, min_count=5, workers=4)
```

gensim.models.Word2Vec

- To initialize a `Word2Vec` model we need to set the following parameters:
  - `size`: number of dimensions
  - `min_count`: tells the model to ignore words with total count less than this number
  - `iter`: number of iterations over the corpus
  - `seed`: to ensure same results are reproduced

```
model = Word2Vec(NYT_clean, size = 50, min_count = 3, iter = 15, seed = 2)
```

```
print(model.vector_size)
```

```
50
```

```
print(NYT_clean[0])
```

```
['pakistan', 'struggling', 'batsmen', 'must', 'find', 'way', 'handle', 'south', 'africa', 'potent',
'pace', 'attack', 'claw', 'way', 'back', 'series', 'second', 'test', 'starts', 'likely', 'lively',
'newlands', 'wicket', 'thursday']
```

Word2Vec: most similar words

- `.wv.most_similar()` is used to find the top ten words similar to the input
- Let's try to find similar words for *government*:

```
print(model.wv.most_similar('government'))
```

```
[('former', 0.8843989968299866), ('latest', 0.8808625936508179), ('friday', 0.8764451146125793),
('back', 0.8755346536636353), ('border', 0.8715096116065979), ('house', 0.8666701912879944),
('tuesday', 0.8539848327636719), ('security', 0.8527489900588989), ('china', 0.8526856899261475),
('new', 0.8490263223648071)]
```

- **Note:**
  - Since Word2Vec has been trained on a generalized context, the results for similar words may not be always accurate
  - In such cases where specific problems of interest need to be solved, we may have to create our own Word Embeddings by training the model on our own corpus of documents

Word2Vec: most similar words

- Now let's try the same for *trade*:

```
print(model.wv.most_similar('trade', topn = 5))
```

```
[('much', 0.8864331245422363), ('strategies', 0.8734394907951355), ('year', 0.8724113702774048), ('de',
0.8623210787773132), ('official', 0.8582102656364441)]
```

- These are the top 5 words closest in meaning to *trade* in our **50-dimensional space**
- By looking at these words, why do you think they came up as most similar?
- What do you think in terms of context of these words?
- What are the limitations of this approach?

Knowledge check 2

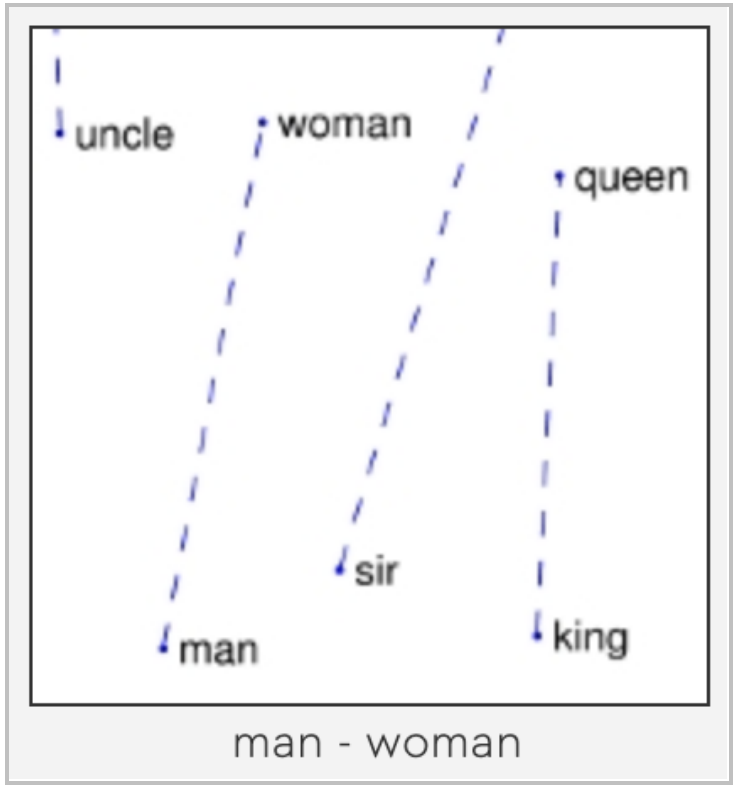
Exercise 1

Module completion checklist

Objective	Complete
Summarize feature engineering and word embeddings	✔
Create a Gensim Word2vec model to view similar words	✔
Download and load GloVe word embeddings	
Use pre-trained word embeddings to generate text features for model development	
Recap cosine similarity and how it applies to our document embeddings matrix	
Compute cosine similarity and find similar documents	

GloVe

- ***GloVe (Global Vectors for Word Representation)*** is an *unsupervised learning algorithm for generating word embeddings* developed by ***the Stanford NLP Group***
- It aggregates a global word-to-word co-occurrence matrix from large text corpora (e.g. Wikipedia, Common Crawl, Twitter)
- The vectors obtained show hidden linear substructures in the vector space that the words represent



GloVe: conceptual model

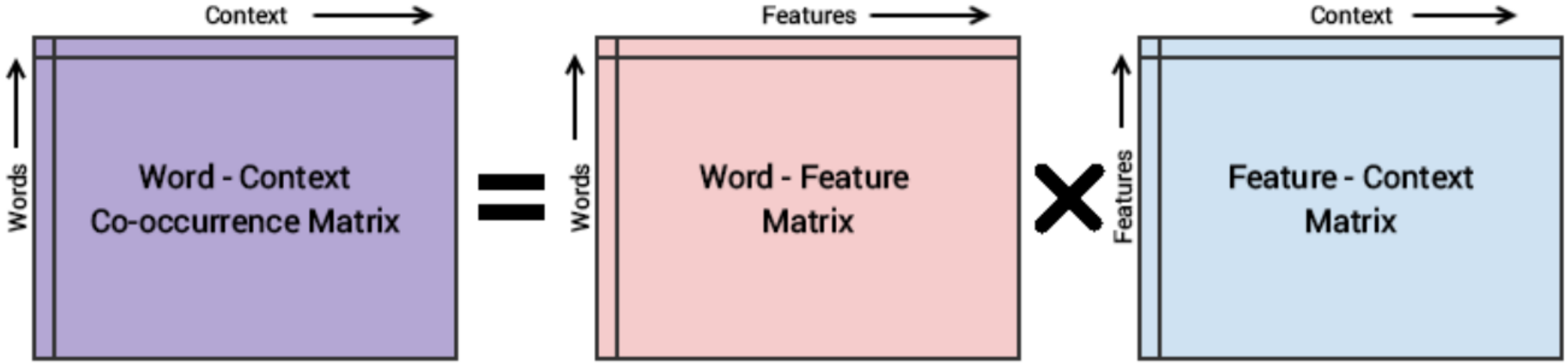
- It constructs a **word context co-occurrence matrix** using statistics across a large text corpus, like English Wikipedia, for instance
  - The matrix consists of word-context pairs, where **each element represents how often words (the rows) appear within the context (the columns)**

I love dogs.
I love cats.
I hate rats.

	I	love	dogs	cats	hate	rats	.
I	0	2	0	0	1	0	0
love	2	0	1	1	0	0	0
dogs	0	1	0	0	0	0	1
cats	0	1	0	0	0	0	1
hate	1	0	0	0	0	1	0
rats	0	0	0	0	1	0	1
.	0	0	1	1	0	1	0

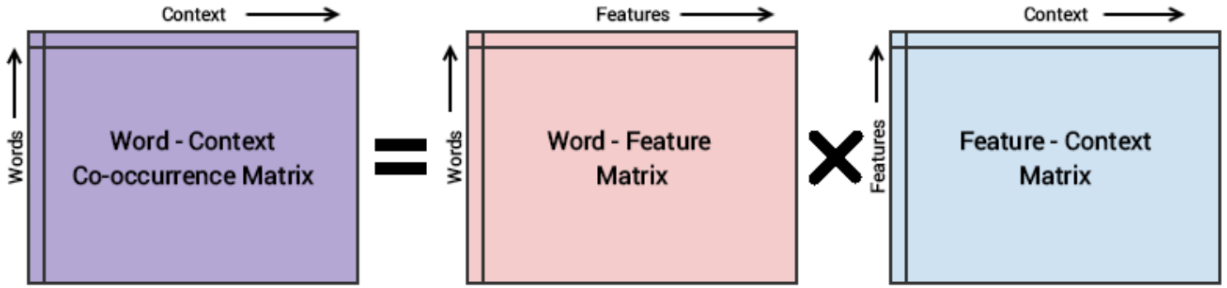
GloVe: conceptual model

- Consider the matrices Word-Context ($\boldsymbol{WC}$) matrix, Word-Feature ($\boldsymbol{WF}$) matrix and Feature-Context ($\boldsymbol{FC}$) matrix
- We try to factorize $\boldsymbol{WC = WFxFC}$ such that $\boldsymbol{WC}$ is reconstructed by multiplying $\boldsymbol{WF}$ and $\boldsymbol{FC}$
- Random weights are assigned to $\boldsymbol{WF}$ and $\boldsymbol{FC}$ and then multiplied to get $\boldsymbol{WC'}$ (an approximation to WC), which is then measured how close it is to $\boldsymbol{WC}$



GloVe: conceptual model

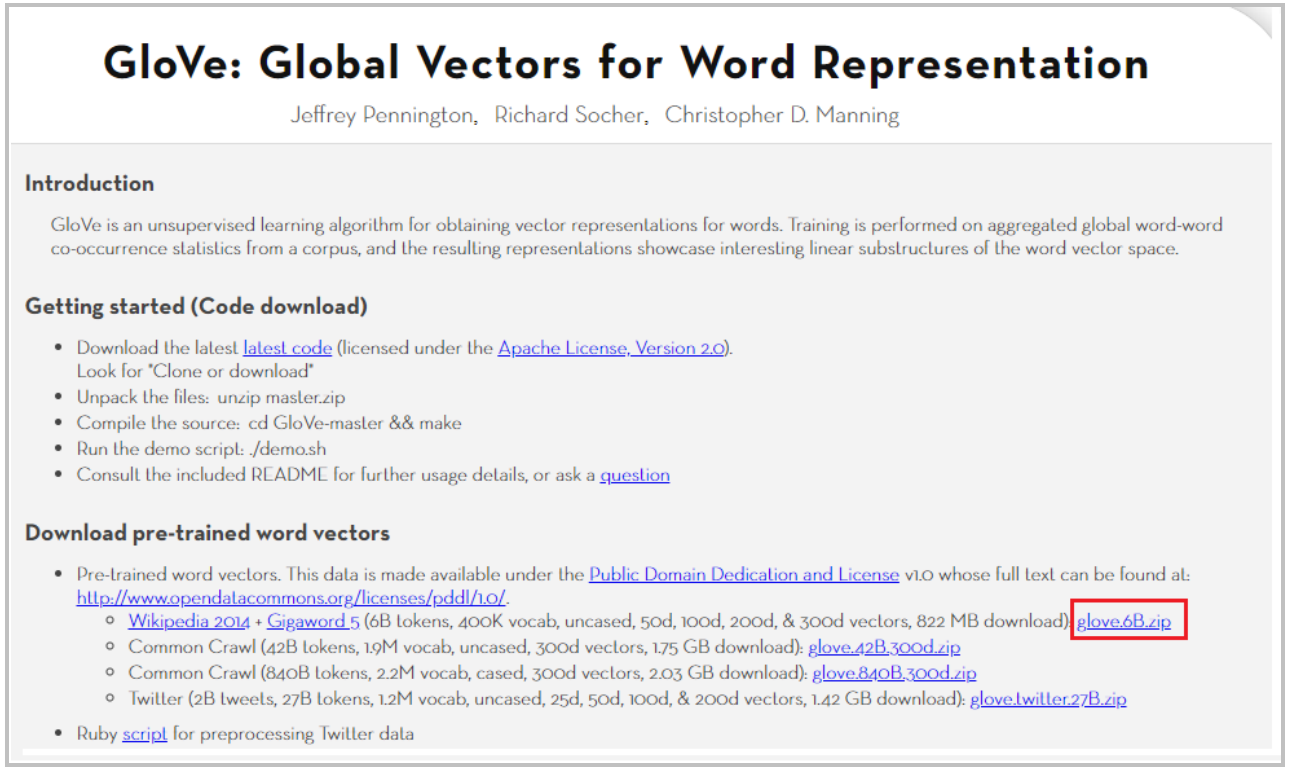
- This is done **multiple times** to adjust the weights and reduce the error
- Finally, the Word-Feature ($W F$) matrix gives us the word embeddings for each word where F is the number of dimensions
- The result is a **learning model that results in better word embeddings**



Source

Download pre-trained word vectors

- You can download the file ***here***
  - `glove.6B.zip` consists of Pre-trained word vectors built using Wikipedia and English newswire data
- Since there are several corpora listed, you would pick the corpus that represents the text closest to the domain you need for your data



GloVe: Global Vectors for Word Representation

Jeffrey Pennington, Richard Socher, Christopher D. Manning

Introduction

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

Getting started (Code download)

- Download the latest latest code (licensed under the Apache License, Version 2.0). Look for "Clone or download"
- Unpack the files: unzip master.zip
- Compile the source: cd GloVe-master && make
- Run the demo script: ./demo.sh
- Consult the included README for further usage details, or ask a question

Download pre-trained word vectors

- Pre-trained word vectors. This data is made available under the Public Domain Dedication and License v1.0 whose full text can be found at: http://www.opendatacommons.org/licenses/pddl/1.0/.
  - Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d, 200d, & 300d vectors, 822 MB download): glove.6B.zip
  - Common Crawl (42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download): glove.42B.300d.zip
  - Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download): glove.840B.300d.zip
  - Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 25d, 50d, 100d, & 200d vectors, 1.42 GB download): glove.twitter.27B.zip
- Ruby script for preprocessing Twitter data

Load GloVe text files

- The glove file has embedding vector sizes: 50, 100, 200, and 300
- There is no hard rule for choosing the number of dimensions - it is largely based on experience of having tried different dimensions on different corpus
- **We will go ahead with 200 dimensions**

```
# Number of glove dimensions.
GLOVE_DIM = 200

# Load pre-trained glove embeddings.
glove_file = data_dir + "/glove.6B.200d.txt"
```

Load GloVe text files

- `LoadGloveModel()` is a function defined to extract word embeddings from the glove file

```
def LoadGloveModel(glove_file):
    print("Loading GloVe Model")
    f = open(glove_file,'r',encoding="utf8")
    model = {}
    for line in f:
        splitLine = line.split()
        word = splitLine[0]
        embedding = np.array([float(val) for val in splitLine[1:]])
        model[word] = embedding
    print("Done.", len(model), " words loaded!")
    return model

# Load embeddings from file.
glove_model = LoadGloveModel(glove_file)
```

```
Loading GloVe Model
Done. 400000  words loaded!
```

Load GloVe text files

- Check the first few embeddings of `glove_model`

```
dict(list(glove_model.items())[0:3])
```

```
{'the': array([-7.1549e-02,  9.3459e-02,  2.3738e-02, -9.0339e-02,  5.6123e-02,
        3.2547e-01, -3.9796e-01, -9.2139e-02,  6.1181e-02, -1.8950e-01,
        1.3061e-01,  1.4349e-01,  1.1479e-02,  3.8158e-01,  5.4030e-01,
       -1.4088e-01,  2.4315e-01,  2.3036e-01, -5.5339e-01,  4.8154e-02,
        4.5662e-01,  3.2338e+00,  2.0199e-02,  4.9019e-02, -1.4132e-02,
        7.6017e-02, -1.1527e-01,  2.0060e-01, -7.7657e-02,  2.4328e-01,
        1.6368e-01, -3.4118e-01, -6.6070e-02,  1.0152e-01,  3.8232e-02,
       -1.7668e-01, -8.8153e-01, -3.3895e-01, -3.5481e-02, -5.5095e-01,
       -1.6899e-02, -4.3982e-01,  3.9004e-02,  4.0447e-01, -2.5880e-01,
        6.4594e-01,  2.6641e-01,  2.8009e-01, -2.4625e-02,  6.3302e-01,
       -3.1700e-01,  1.0271e-01,  3.0886e-01,  9.7792e-02, -3.8227e-01,
        8.6552e-02,  4.7075e-02,  2.3511e-01, -3.2127e-01, -2.8538e-01,
        1.6670e-01, -4.9707e-03, -6.2714e-01, -2.4904e-01,  2.9713e-01,
        1.4379e-01, -1.2325e-01, -5.8178e-02, -1.0290e-03, -8.2126e-02,
        3.6935e-01, -5.8442e-04,  3.4286e-01,  2.8426e-01, -6.8599e-02,
        6.5747e-01, -2.9087e-02,  1.6184e-01,  7.3672e-02, -3.0343e-01,
        9.5733e-02, -5.2860e-01, -2.2898e-01,  6.4079e-02,  1.5218e-02,
        3.4921e-01, -4.3960e-01, -4.3983e-01,  7.7515e-01, -8.7767e-01,
       -8.7504e-02,  3.9598e-01,  6.2362e-01, -2.6211e-01, -3.0539e-01,
       -2.2964e-02,  3.0567e-01,  6.7660e-02,  1.5383e-01, -1.1211e-01,
       -9.1540e-02,  8.2562e-02,  1.6897e-01, -3.2952e-02, -2.8775e-01,
       -2.2320e-01, -9.0426e-02,  1.2407e+00, -1.8244e-01, -7.5219e-03,
       -4.1388e-02, -1.1083e-02,  7.8186e-02,  3.8511e-01,  2.3334e-01,
        1.4414e-01, -9.1070e-04, -2.6388e-01, -2.0481e-01,  1.0099e-01,
        1.4076e-01,  2.8834e-01, -4.5429e-02,  3.7247e-01,  1.3645e-01,
```

Module completion checklist

Objective	Complete
Summarize feature engineering and word embeddings	✔
Create a Gensim Word2vec model to view similar words	✔
Download and load GloVe word embeddings	✔
Use pre-trained word embeddings to generate text features for model development	
Recap cosine similarity and how it applies to our document embeddings matrix	
Compute cosine similarity and find similar documents	

Word count of corpus

- Let's create a frequency count of each word in the corpus and save as a dataframe
- We will need this to create a word embedding matrix in the next step

```
# Save series as a dictionary.
corpus_freq_dist = DTM_not_stemmed.sum(axis = 0).to_dict()
dict(list(corpus_freq_dist.items())[0:5])
```

```
{'abducted': 1, 'able': 1, 'abo': 1, 'absentee': 1, 'abuse': 1}
```

```
# Extract word counts for exploratory analysis.
word_counts = pd.DataFrame(list(corpus_freq_dist.items()), columns = ['word', 'count'])
```

```
print(word_counts.head())
```

```
        word  count
0  abducted      1
1      able      1
2       abo      1
3  absentee      1
4     abuse      1
```

Word embeddings matrix

```
# Initialize embeddings matrix.
DICT_SIZE = len(word_counts.index)
word_emb_matrix = np.zeros((DICT_SIZE, GLOVE_DIM))
words = list(word_counts.word)
NUM_MESSAGES = len(NYT_tokenized)
```

- The loop will extract the vector representation of each word found in the dictionary and save to a word embedding matrix

```
for i in range(DICT_SIZE):
    w = words[i]
    vect = glove_model.get(w)

    if vect is not None:
        word_emb_matrix[i] = vect
```

Word embeddings matrix

- The result matrix contains the word embeddings having 200 dimensions of each word (2272 words total) in our corpus

```
print(word_emb_matrix.shape)
```

```
(2272, 200)
```

```
print(word_emb_matrix[0])
```

```
[-0.43671   -0.20781   -0.01403   -0.28693   -0.093612  -0.25447
  0.079885   0.74519   -0.11048   -0.32408    0.27374   -0.19368
  0.33044    0.31688   -0.5451    -0.29608    0.78748    0.26771
 -0.4236     0.60147    0.14603    1.9923    -0.46951   -0.59057
  0.33249   -0.37852   -0.18481   -0.82489   -0.067537   0.62164
 -0.29394    0.51388    0.20377   -0.025748   0.83196   -0.2673
 -0.33103   -0.080671  -0.20598    0.56917   -0.065674  -0.16056
 -0.13943   -0.40235   -0.22773   -0.29537   -0.19651   -0.91085
  0.5316    -0.13412   -0.030422   0.49585   -0.055606  -0.50374
 -0.7382     0.29312   -0.55273   -0.6539     0.86494    0.0033696
  0.29673    0.34039    0.22361    0.6434     0.2151    -0.6501
  1.1063    -0.7135    -0.63416   -0.083591  -0.62089    0.37116
  0.27629    0.6649     0.24002    0.14323    0.34324    0.2507
  0.53446   -0.11818   -0.22189    0.51611   -0.71737   -0.14709
  0.33113   -0.0401     0.13478   -0.18695   -1.3606     0.74616
  0.2637     0.18546   -0.28705   -0.25314   -0.9239     0.58111
 -0.33172    0.2827     0.62193   -0.027336   0.38448    0.021043
```

NYT Article embeddings matrix

- The next step is to create a **(document) embedding matrix**, where each document would be represented as numeric vectors
- Multiply the DTM by the embedding matrix to produce the sum of all vectors for words found in the snippets

```
# Convert dataframe to a numpy array.
DTM_not_stemmed = DTM_not_stemmed.to_numpy()

# Compute sums of all word counts for each chat message.
DTM_row_sums = np.sum(DTM_not_stemmed, axis=1)

NYT_embeddings_matrix = DTM_not_stemmed.dot(word_emb_matrix)
print(DTM_not_stemmed.shape)
```

```
(250, 2272)
```

NYT Article embeddings

- Average the snippet embeddings of a document by the total count of words in that document
- We do so to get a weighted average of that document in the corpus

```
for i in range(NUM_MESSAGES):
    NYT_embeddings_matrix[i] = np.true_divide(NYT_embeddings_matrix[i], DTM_row_sums[i])
```

```
# Save as a dataframe and add NYT snippet IDs.
NYT_emb_df = pd.DataFrame(NYT_embeddings_matrix)
```

```
print(NYT_emb_df.head())
```

```
         0         1         2    ...       197       198       199
0  0.064769  0.252212 -0.337972   ...  0.126306 -0.169112  0.048444
1 -0.186589 -0.003260 -0.428050   ...  0.092167  0.033890  0.299560
2 -0.042506  0.182100 -0.506429   ...  0.174178  0.018378  0.120248
3  0.183803  0.244767 -0.040656   ...  0.145590  0.122590  0.143893
4 -0.230828  0.192996 -0.278713   ... -0.125005 -0.026874  0.122185

[5 rows x 200 columns]
```

Knowledge check 3

Exercise 2

Module completion checklist

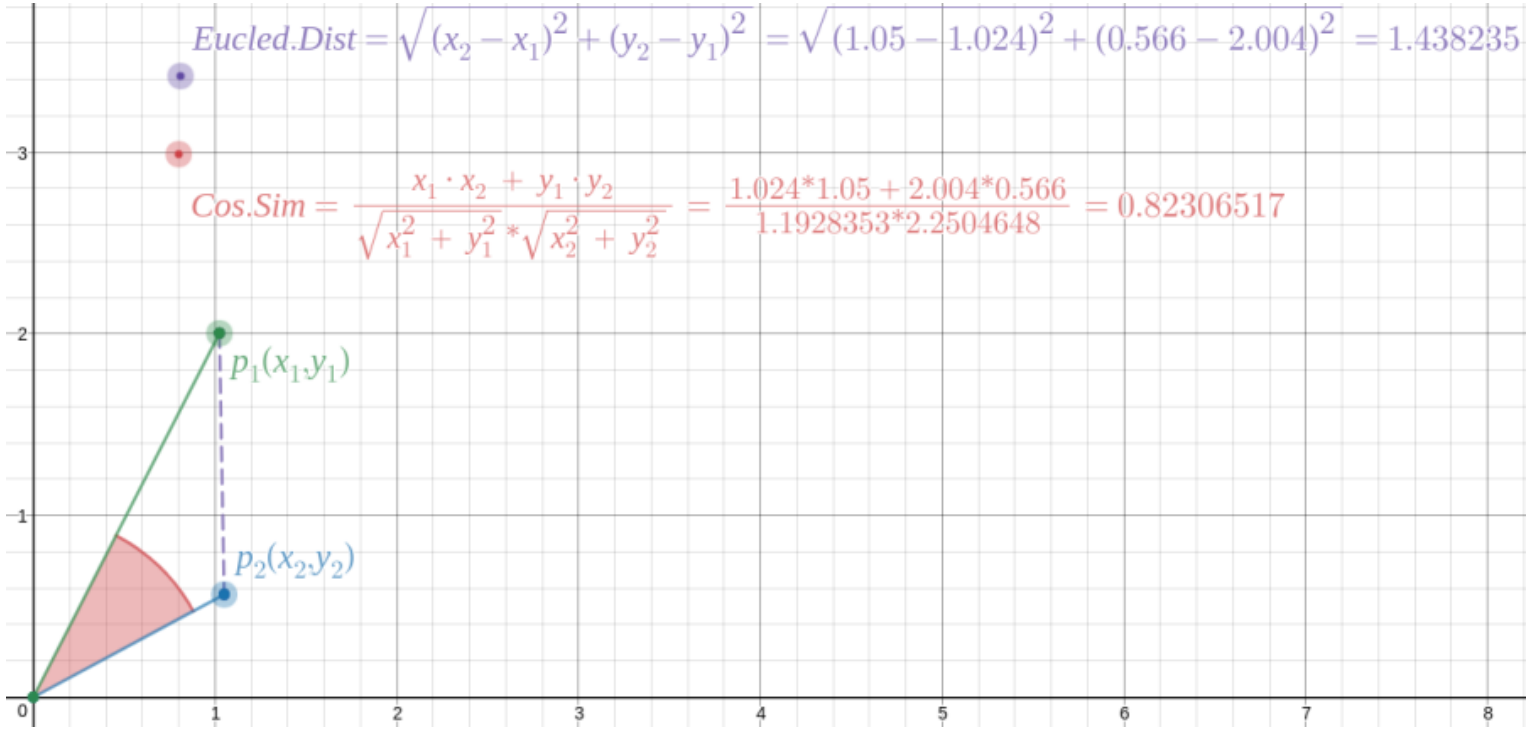
Objective	Complete
Summarize feature engineering and word embeddings	✔
Create a Gensim Word2vec model to view similar words	✔
Download and load GloVe word embeddings	✔
Use pre-trained word embeddings to generate text features for model development	✔
Recap cosine similarity and how it applies to our document embeddings matrix	
Compute cosine similarity and find similar documents	

Measuring similarity in text: recap

- When measuring similarity between 2 data points, the usual metric would be ***Euclidean distance***
- When working with text data, we often use a metric called **cosine similarity**

Recap: cosine similarity

- **Cosine similarity** is $cos(a)$, where a is an angle between the two *vectors*
- These vectors are either *terms* or *documents*, depending on what we would like to compare
- In order **to compute cosine similarity between documents, we need to know their numeric representation**, which we already have in the form of `corpus_tfidf`

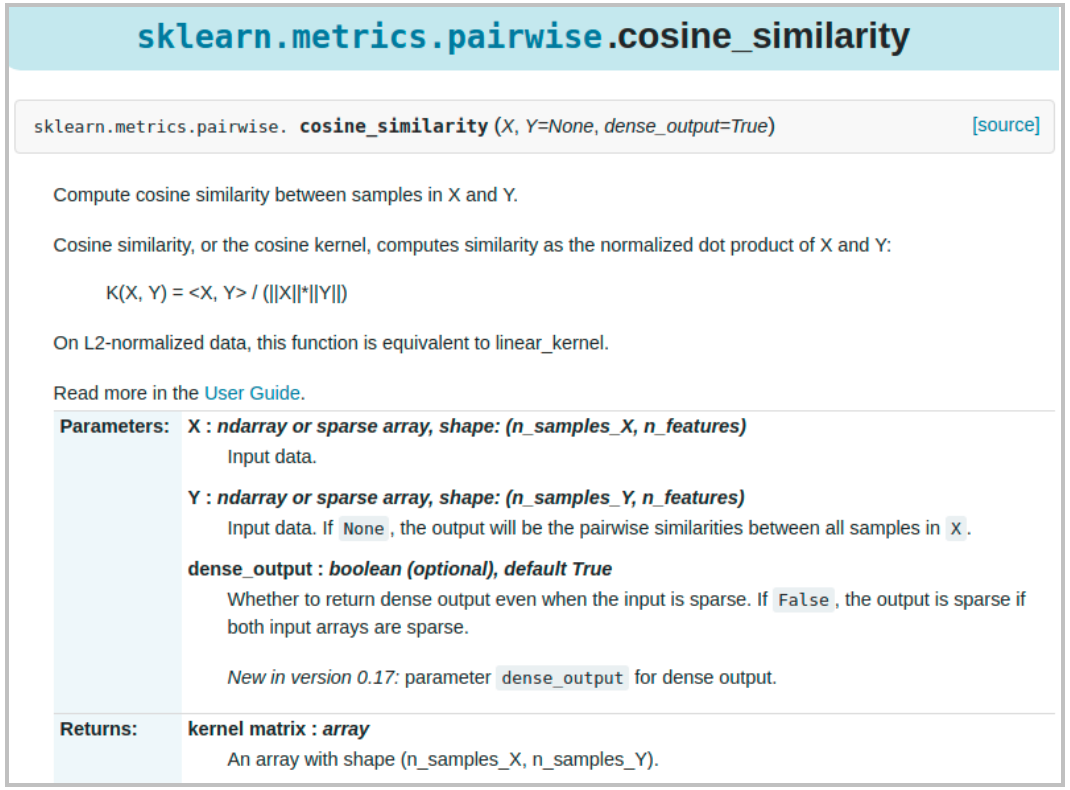


Module completion checklist

Objective	Complete
Summarize feature engineering and word embeddings	✔
Create a Gensim Word2vec model to view similar words	✔
Download and load GloVe word embeddings	✔
Use pre-trained word embeddings to generate text features for model development	✔
Recap cosine similarity and how it applies to our document embeddings matrix	✔
Compute cosine similarity and find similar documents	

Recap: cosine similarity with scikit-learn

- We'll use `cosine_similarity` to compute cosine similarity for each possible pair of documents
- The input for this function is a matrix of row vectors



sklearn.metrics.pairwise.cosine_similarity

`sklearn.metrics.pairwise.` **`cosine_similarity`** (*X*, *Y=None*, *dense_output=True*) [source]

Compute cosine similarity between samples in X and Y.

Cosine similarity, or the cosine kernel, computes similarity as the normalized dot product of X and Y:

K(X, Y) = <X, Y> / (||X||*||Y||)

On L2-normalized data, this function is equivalent to linear_kernel.

Read more in the User Guide.

Parameters:

X : ***ndarray or sparse array, shape: (n_samples_X, n_features)***
Input data.

Y : ***ndarray or sparse array, shape: (n_samples_Y, n_features)***
Input data. If `None`, the output will be the pairwise similarities between all samples in `X`.

dense_output : ***boolean (optional), default True***
Whether to return dense output even when the input is sparse. If `False`, the output is sparse if both input arrays are sparse.

New in version 0.17: parameter `dense_output` for dense output.

Returns:

kernel matrix : ***array***
An array with shape (n_samples_X, n_samples_Y).

Find documents similar to snippet 25

- Take a look at the NYT message below

```
NYT_snippets = NYT['snippet']
NYT_snippets[1]
```

```
'The National Football League is under the microscope for lack of minority head coaches after a recent
slew of firings in the league.    '
```

- Suppose you wish to see similar snippets as this one, we can run cosine similarity between this snippet and the NYT snippets represented by the embedding matrix
- We can find the most similar vectors based on their similarity scores

Find documents similar to snippet 25

- We will save the vector of NYT snippet `25` to another variable
- It is of type pandas series, hence we need to convert it to a numpy array to run mathematical operations on it

```
# Average embeddings.
target_NYT_emb =
NYT_emb_df.loc[1].to_numpy()
target_NYT_emb[0:5]
```

```
array([-0.18658917, -0.00326015,
-0.42805033,  0.13581196,
0.00246592])
```

- To get similarity scores for each document, we need to reshape array `target_NYT_emb` to a single array
- Adding `-1` to the `reshape()` functions makes sure the new shape is compatible to the original array and the result will be a 1-D array of that length

```
target_NYT_emb.reshape(1, -1)
```

```
array([[-1.86589167e-01, -3.26015000e-03, -4.28050333e-01,
         1.35811958e-01,  2.46591667e-03,  1.98897583e-01,
        -3.18630833e-01,  2.86251483e-01,  2.84825100e-01,
         8.02603833e-02,  2.65277158e-01,  7.05543333e-02,
        -5.43455167e-02, -3.19347500e-02,  5.81319833e-01,
         1.36398250e-01, -1.54404667e-01,  2.29646083e-01,
        -2.18793333e-01, -2.20230000e-02,  2.84038417e-01,
         2.02163667e+00, -4.17531750e-02, -3.03185083e-01,
         3.49049583e-01,  1.42592525e-01,  2.16960733e-01,
        -2.57965000e-02,  1.63046417e-01, -2.40800500e-01,
         1.39037167e-01, -1.44992083e-01, -3.92232500e-02,
        -1.18369167e-02, -2.10918417e-01, -2.21815000e-01,
        -3.83510417e-01, -2.28433333e-02,  2.69130333e-01,
         2.54149167e-01,  7.10660417e-02, -1.57577633e-01,
         3.57017083e-01,  1.06266000e-01, -8.32066667e-02,
         1.01090917e-01, -2.62902500e-02,  7.04259167e-02,
        -1.09815250e-01,  6.69086667e-02,  9.67110833e-02,
```

Compute cosine similarity

```
similarity_scores = cosine_similarity(NYT_emb_df, target_NYT_emb.reshape(1, -1))
similarity_scores[0:5]
```

```
array([[0.67984934],
       [1.        ],
       [0.75331666],
       [0.57959795],
       [0.67846477]])
```

```
similarity_scores_df = pd.DataFrame(similarity_scores,
                                    columns = ['similarity_score'],
                                    index = NYT.index)
print(similarity_scores_df.head())
```

```
   similarity_score
0          0.679849
1          1.000000
2          0.753317
3          0.579598
4          0.678465
```

View results

- `sort_values()` will sort the similarity scores by value in descending order

```
similarity_scores_df.sort_values('similarity_score', ascending = False).head()
```

```
     similarity_score
1            1.000000
142          1.000000
168          0.820886
112          0.794599
9            0.780351
```

- Why do you think snippets 1 and 142 has a perfect 1.0 score?
  - Because the documents are perfectly similar! In other words, they are the same document
- This is a good way to keep sanity checks and see if our code makes sense
- Let's take a look at the similar snippets

View results

```
print(NYT_snippets[168])
```

```
The coach, who served a suspension earlier this season, had already announced his retirement, but he coached his team to a win over Washington in his final game.
```

```
print(NYT_snippets[112])
```

```
The competition’s third round, when the Premier League’s giants join the lower-level dreamers, has been devalued by disinterest. But there’s still joy if you know where to look.
```

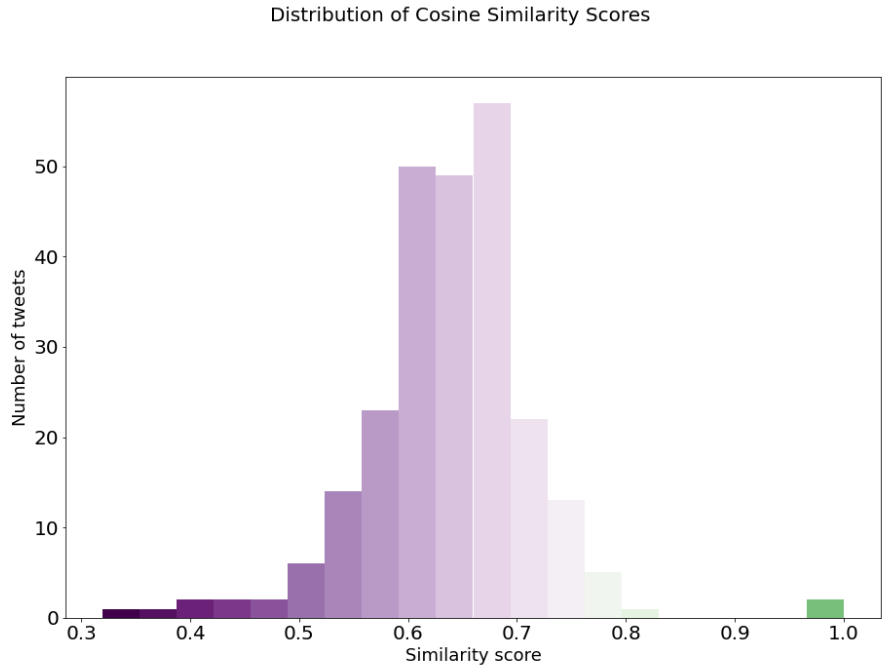
- These results are similar to the input snipped as they all fall under **sports** category

Cosine similarity score distribution plot

- A histogram of `similarity_score` will show us its distribution
- We can see that its frequency is high between `0.8` and `1.0`

```
# Plot results.
fig = plt.figure(figsize=(15,10))
cm = plt.cm.PRGn
n, bins, patches =
plt.hist(similarity_scores_df['similarity_score'],

                              20, color='green')
for i, p in enumerate(patches):
    plt.setp(p, 'facecolor', cm(i/25)) # notice
the i/25
fig.suptitle('Distribution of Cosine Similarity
Scores', fontsize=20)
plt.xlabel("Similarity score", fontsize=18)
plt.ylabel("Number of NYT snippets",
fontsize=18)
plt.show()
```



Knowledge check 4

Exercise 3

Module completion checklist

Objective	Complete
Summarize feature engineering and word embeddings	✔
Create a Gensim Word2vec model to view similar words	✔
Download and load GloVe word embeddings	✔
Use pre-trained word embeddings to generate text features for model development	✔
Recap cosine similarity and how it applies to our document embeddings matrix	✔
Compute cosine similarity and find similar documents	✔

This completes our module

Congratulations!

Wondering what's next?

- You may be interested in the following course, which Data Society offers for Booz Allen Hamilton:
- **Text Mining Capstone Workshop** - In this hands-on workshop, students will work closely alongside an expert instructor to design, develop, and deploy a text mining and NLP project of the student's choice. This program will provide students access to an environment where they can set time, ask questions, receive regular instructor support, and work toward creating a bespoke solution. Students will bolster their portfolio with a project that showcases their ability to apply cutting-edge techniques to real-world data.