CITS4012 Natural Language Processing

A/Prof. Wei Liu

Sep 07, 2021

CONTENTS

This Jupyter Book hosts the **lab materials** of CITS4012 Natural Language Processing offered at Department of Computer Science and Software Engineering at the University of Western Australia.

All other teaching materials of the unit can be found at UWA LMS - The University of Western Australia.

CHAPTER

ONE

LAB 01: CONDA ENVIRONMENT AND PYTHON REFRESHER

Welcome to CITS4012 Natural Language Processing. NLP is a very practical discipline that requires familiarity with Python, Numpy, Pandas and many similar yet distinctive text processing packages. So let's started by getting used to handle multiple environments as often different NLP toolkits have conflict requirement on package or Python versions. Then we will move on with some refresher of Python and Numpy.

1.1 Google Colab

For quick code testing without worrying too much about installing packages, one can use Google Colaboratory, "Colab" for short. If you are not familar with Colab, please follow the intructions to either create a Jupyter Notebook or look through the sample ones.

1.2 Bring you own device

The following two pages contain instructions on how to set up multiple "isolated" Python environments on your own computer. This is the recommended mode of study because as system adminstrator you have a lot more control on what you can install and when, and the kernerls will not disconnect with a time limit like Google Colab. Fluency in managing multiple environments is also a desired learning outcome of this unit. So despite it can be frustrating and tedious at times, it is rewarding to manage your own coding environment. Please do remember to make use of repositories such as Github for regular backups.

1. Setting up CITS4012 Base environment
2. Setting up CITS4012 MISC environment

1.3 Use Lab Machines

1. Use Lab Machines

1.4 Python Refresher

1. Python Basics
2. Iterables
3. Numpy (for math and matrix operations)
4. Matplotlib (for plotting)

Original Refresher Notebooks credit to Stanford 224n

1.5 CITS4012 Base Environment

1.5.1 IDEs and Git

Installing Visual Studio Code

- Visual Studio Code Download
- 64 bit System Installer (`VSCodeSetup-x64-1.57.1.exe` - 78KB)
- Default Installation Path (`C:\Program Files\Microsoft VS Code`)
- Installation Time (~5 mins)

Installing Extensions for Visual Studio Code

- Python Extension for VSCode Instruction
- Remote - SSH Extension
- Remote X11 (SSH) Extension

Install git

- Latest Version of Git for Windows
- Don't forget to add git to the system PATH

1.5.2 Conda and Unit Specific Packages

Installing Anaconda

- Anaconda Installation Instructions
- Individual Version (`Anaconda3-2021.05-Windows-x86_64.exe` - 488KB)
- Installation Size (2.9 GB)
- Installed for all users at `C:\ProgramData\Anaconda3`
- Installation Time (~30 mins)
- Launch Anaconda Navigator

You can start a CMD or POWERSHELL console using the navigator, or following the steps 1 and 2 in the screenshot below to start a CMD or POWERSHELL. If you are intalling packages, you can right click the arrow to bring up a pop-up menu, `run as adminstrator` (Step 3a) or `pin on taskbar` (Step 3b) for future convenience.

Install from an environment YAML file

If we install from this YAML file, then we can ignore all the following steps after this section.

First download the installation file here: cits4012.yml

```
conda env create -p c:\envs\cits4012 --file cits4012.yml
```

If this is successful, you can ignore the rest of the steps below.

Install packages step by step from scratch

Create Anaconda Environment

Need admin access to write to C drive (Run Conda Powershell as Administrator - right click on the icon)

```
conda create -p c:\envs\cits4012 python=3.8
conda activate c:\envs\cits4012
```

Use the virtual environment in VSCode

- Instructions on how to use environment in VSCode
- Test to see if the CITS4012_base environment is available from VSCode

Install Spacy

1. Go back to Conda CMD.exe, check to see if you have `pip` installed using

```
conda list
pip install -U spacy
python -m spacy download en_core_web_sm
```

1. Find the Spacy version (we want v3+):

```
# Windows CMD
C:\> conda list | findstr "spacy"`

# Windows PowerShell
C:\> conda list | Select-String "spacy"

# Linux
$ conda list | grep "spacy"
```

Install PyTorch

Check for Cuda compatible Graphics Card on Windows

1. Click Start.
2. On the Start menu, click Run.
3. In the Open box, type “dxdiag” (without the quotation marks), and then click OK.
4. The DirectX Diagnostic Tool opens. …
5. On the Display tab, information about your graphics card is shown in the Device section.

My laptop has NVIDIA GeForce MX130.

Install Pytorch

Pytorch Website

- with GPU

```
conda install pytorch torchvision torchaudio torchtext cudatoolkit=11.1 -c pytorch -c␣
↪conda-forge
```

- CPU only

```
conda install pytorch torchvision torchaudio torchtext cpuonly -c pytorch -c conda-
↪forge
```

Install Tensorboard

```
conda install -c conda-forge tensorboard
```

Install GraphViz on Windows

2.47.3 EXE installer for Windows 10 (64-bit)

Download the exe file and install, make sure it is added to the system PATH (Windows - Edit the Windows Environment Variables.

Install torchviz

```
pip install torchviz
```

Install NLTK

```
pip install nltk
```

and then download the data and models

```
python -m nltk.downloader -d c:\envs\cits4012\nltk_data all
```

Install truecase

install this after NLTK installation pls.

```
pip install truecase
```

Install Jupyterlab

```
conda install -c conda-forge jupyterlab
```

Install Scikit-learn

```
pip install -U scikit-learn
```

Verify if it works:

```
python -c "import sklearn; sklearn.show_versions()"
```

Install Matplotlib

```
pip install matplotlib
```

Install gensim

Note this will install the latest gensim 4.x

```
conda install -c conda-forge gensim
```

Install import-ipynb

This is a package to import functions from other Jupyter Notebooks.

```
pip install import-ipynb
```

Install Python Levenshtein Similarity

```
pip install python-Levenshtein
```

Install interactive visualisation bokeh

```
pip install bokeh
```

Install bs4 to get BeautifulSoup for web crawling

```
pip install bs4
```

Finally Export Environment into an YAML file

```
conda env export -p c:\envs\cits4012 --no-builds -f cits4012.yml
```

1.6 CITS4012 MISC Enviornment

1.6.1 Frequently encounted problems in Windows

1. File not found when files are actually there - this might be to do with the limitation of path length (<=260 characters) in Windows. See below for a solution

- How to set group policies in Windows to allow for long path

1. When trying to install some pip packages you may get the error stating:

   Microsoft Visual C++ 14.0 is required. Get it with "Build Tools for Visual Studio": https://visualstudio.microsoft.com/downloads/

Scroll down the page to find "All Tools", expand "Tools for Visual Studio 2019" and find "Build Tools for Visual Studio 2019", download and install as an admin for all users.

Warning: The installation of the Build Tools might take a while (~20mins).

1.6.2 Install from an environment YAML file

If we install from this YAML file, then we can ignore all the following steps after this.

Download the environment YAML file here: cits4012_py37.yml

```
conda env create -p c:\envs\cits4012_py37 --file cits4012_py37.yml
```

If this is successful, you can ignore the rest of the steps below.

1.6.3 Install step by step

Create a new environment

```
conda create -p c:\envs\cits4012_py37 python=3.7
conda activate c:\envs\cits4012_py37
```

Install Flair

Flair requires different versions of numpy and torch, so it is better to isolate it from the normal environment

```
pip install flair
```

Install Neuralcoref

Follow the "compile from source instruction" on the github page as it requires Python 3.7 and Spacy 2.0+

```
git clone https://github.com/huggingface/neuralcoref.git
cd neuralcoref
pip install -r requirements.txt
pip install -e .
```

You may run into the frequent problem 2 above. Solve it by installing C++ compiler suitable for your OS.

Install Jupyterlab

```
pip install jupyterlab
```

Install Matplotlib

```
pip install matplotlib
```

Export Environment Files

```
conda env export -p c:\envs\cits4012_py37 --no-builds -f cits4012_py37.yml
```

1.7 Use Lab Machines

Our labs support dual boot into either Window 10 or Redhat 7. Depending on which OS you prefer, you can follow the below instructions to activate the environment. However, please note these are *un-writable environments*. It is not possible to install new packages during the semester, as our IT team maintains a university-wide image. Installation during semester may disturb the global computing environment.

It is useful if you want some spare CPUs to train your model as the lab machines have decent CPU processing power, while carrying out other activities on your own devices.

Windows

1. Open the start menu and type 'Anaconda Prompt' into the search box.
2. Open 'Anaconda Prompt (Anaconda3)' or 'Anaconda Powershell Prompt (Anaconda3)'
3. Type `conda activate A:\Anaconda\envs\cits4012` for the 'cits4012' environment or `conda activate A:\Anaconda\envs\cits4012_py37` to activate the 'cits4012_py37' environment and press enter.

Red Hat Linux

1. Select 'Applications' in the top left corner and select 'CSSE' -> 'Anaconda Prompt'
2. In terminal, type `conda activate ~/anaconda/envs/cits4012` to activate the 'cits4012' environment or `conda activate ~/anaconda/envs/cits4012_py37` to activate the 'cits4012_py37' environment and press enter.

1.8 Python Basics

https://www.w3schools.com/python/

```
# input and output
name = input()
print("hello, " + name)
```

```
hello, CITS4012 Natural Language Processing
```

```
# print multiple variables separated by a space
print("hello", name, 1, 3.0, True)
```

```
hello CITS4012 Natural Language Processing 1 3.0 True
```

```
# line comment
"""
block
comments
"""
```

```
'\nblock \ncomments\n'
```

```
# variables don't need explicit declaration
var = "hello" # string
var = 10.0    # float
var = 10      # int
var = True    # boolean
var = [1,2,3] # pointer to list
var = None    # empty pointer
```

```
# type conversions
var = 10
print(int(var))
print(str(var))
print(float(var))
```

```
10
10
10.0
```

```
# basic math operations
var = 10
print("var + 4 =", 10 + 4)
print("var - 4 =", 10 - 4)
print("var * 4 =", 10 * 4)
print("var ^ 4=", 10 ** 4)
print("int(var) / 4 =", 10//4)    # // for int division
print("float(var) / 4 =", 10/4)   # / for float division
# All compound assignment operators available
# including += -= *= **= /= //=
# pre/post in/decrementers not available (++ --)
```

```
var + 4 = 14
var - 4 = 6
var * 4 = 40
var ^ 4= 10000
int(var) / 4 = 2
float(var) / 4 = 2.5
```

```
# basic boolean operations include "and", "or", "not"
print("not True is", not True)
print("True and False is", True and False)
print("True or False is", True or False)
```

```
not True is False
True and False is False
True or False is True
```

```
# String operations
# '' and "" are equivalent
s = "String"
#s = 'Mary said "Hello" to John'
#s = "Mary said \"Hello\" to John"

# basic
print(len(s)) # get length of string and any iterable type
print(s[0]) # get char by index
print(s[1:3]) # [1,3)
print("This is a " + s + "!")

# handy tools
print(s.lower())
print(s*4)
print("ring" in s)
print(s.index("ring"))

# slice by delimiter
print("I am a sentence".split(" "))
# concatenate a list of string using a delimiter
print("...".join(['a','b','c']))

# formatting variables
print("Formatting a string like %.2f"%(0.12345))
print(f"Or like {s}!")
```

```
6
S
tr
This is a String!
string
StringStringStringString
True
2
['I', 'am', 'a', 'sentence']
a...b...c
Formatting a string like 0.12
Or like String!
```

```
# control flows
# NOTE: No parentheses or curly braces
#       Indentation is used to identify code blocks
#       So never ever mix spaces with tabs
for i in range(0,5):
    for j in range(i, 5):
        print("inner loop")
    print("outer loop")
```

```
# if-else
var = 10
if var > 10:
    print(">")
elif var == 10:
    print("=")
else:
    print("<")
```

```
=
```

```
# use "if" to check null pointer or empty arrays
var = None
if var:
    print(var)
var = []
if var:
    print(var)
var = "object"
if var:
    print(var)
```

```
object
```

```
# while-loop
var = 5
while var > 0:
    print(var)
    var -=1
```

```
5
4
3
2
1
```

```
# for-loop
for i in range(3):  # prints 0 1 2
    print(i)

"""
equivalent to
for (int i = 0; i < 3; i++)
"""
print("-------")
# range (start-inclusive, stop-exclusive, step)
for i in range(2, -3, -2):
    print(i)
"""
equivalent to
for (int i = 2; i > -3; i-=2)
"""
```

```
0
1
2
-------
2
0
-2
```

```
'\nequivalent to\nfor (int i = 2; i > -3; i-=2)\n'
```

```
# define function
def func(a, b):
    return a + b
func(1,3)
```

```
4
```

```
# use default parameters and pass values by parameter name
def rangeCheck(a, min_val = 0, max_val=10):
    return min_val < a < max_val    # syntactic sugar
rangeCheck(5, max_val=4)
```

```
False
```

```
# define class
class Foo:

    # optinal constructor
    def __init__(self, x):
        # first parameter "self" for instance reference, like "this" in JAVA
        self.x = x

    # instance method
    def printX(self): # instance reference is required for all function parameters
        print(self.x)

    # class methods, most likely you will never need this
    @classmethod
    def printHello(self):
        print("hello")

obj = Foo(6)
obj.printX()
```

```
6
```

```
# class inheritance - inherits variables and methods
# You might need this when you learn more PyTorch
class Bar(Foo):
    pass
obj = Bar(3)
obj.printX()
```

```
3
```

1.9 Iterables

```
alist = list()  # linear, size not fixed, not hashable
atuple = tuple() # linear, fixed size, hashable
adict = dict()  # hash table, not hashable, stores (key,value) pairs
aset = set()    # hash table, like dict but only stores keys
acopy = alist.copy() # shallow copy
print(len(alist)) # gets size of any iterable type
```

```
0
```

```
# examplar tuple usage
# creating a dictionary to store ngram counts
d = dict()
d[("a","cat")] = 10
d[["a","cat"]] = 11
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-47597361a541> in <module>
      3 d = dict()
      4 d[("a","cat")] = 10
----> 5 d[["a","cat"]] = 11

TypeError: unhashable type: 'list'
```

```
"""
List: not hashable (i.e. can't use as dictionary key)
      dynamic size
      allows duplicates and inconsistent element types
      dynamic array implementation
"""
# list creation
alist = []          # empty list, equivalent to list()
alist = [1,2,3,4,5] # initialized list

print(alist[0])
alist[0] = 5
print(alist)

print("-"*10)
# list indexing
print(alist[0]) # get first element (at index 0)
print(alist[-2]) # get last element (at index len-1)
print(alist[3:]) # get elements starting from index 3 (inclusive)
print(alist[:3]) # get elements stopping at index 3 (exclusive)
print(alist[2:4]) # get elements within index range [2,4)
print(alist[6:]) # prints nothing because index is out of range
print(alist[::-1]) # returns a reversed list

print("-"*10)
# list modification
alist.append("new item") # insert at end
alist.insert(0, "new item") # insert at index 0
alist.extend([2,3,4]) # concatenate lists
```

(continues on next page)

(continued from previous page)

```
# above line is equivalent to alist += [2,3,4]
alist.index("new item") # search by content
alist.remove("new item") # remove by content
alist.pop(0) # remove by index
print(alist)

print("-"*10)
if "new item" in alist:
    print("found")
else:
    print("not found")

print("-"*10)
# list traversal
for ele in alist:
    print(ele)

print("-"*10)
# or traverse with index
for i, ele in enumerate(alist):
    print(i, ele)
```

```
1
[5, 2, 3, 4, 5]
----------
5
4
[4, 5]
[5, 2, 3]
[3, 4]
[]
[5, 4, 3, 2, 5]
----------
[2, 3, 4, 5, 'new item', 2, 3, 4]
----------
found
----------
2
3
4
5
new item
2
3
4
----------
0 2
1 3
2 4
3 5
4 new item
5 2
6 3
7 4
```

```
"""
Tuple: hashable (i.e. can use as dictionary key)
       fixed size (no insertion or deletion)
"""
# it does not make sense to create empty tuples
atuple = (1,2,3,4,5)
 # or you can cast other iterables to tuple
atuple = tuple([1,2,3])

# indexing and traversal are same as list
```

```
"""
Named tuples for readibility
"""
from collections import namedtuple
Point = namedtuple('Point', 'x y')
pt1 = Point(1.0, 5.0)
pt2 = Point(2.5, 1.5)
print(pt1.x, pt1.y)
```

```
1.0 5.0
```

```
"""
Dict: not hashable
      dynamic size
      no duplicates allowed
      hash table implementation which is fast for searching
"""
# dict creation
adict = {} # empty dict, equivalent to dict()
adict = {'a':1, 'b':2, 'c':3}
print(adict)

# get all keys in dictionary
print(adict.keys())

# get value paired with key
print(adict['a'])
key = 'e'

# NOTE: accessing keys not in the dictionary leads to exception
if key in adict:
    print(adict[key])

# add or modify dictionary entries
adict['e'] = 10 # insert new key
adict['e'] = 5  # modify existing keys

print("-"*10)
# traverse keys only
for key in adict:
    print(key, adict[key])

print("-"*10)
# or traverse key-value pairs together
for key, value in adict.items():
```

(continues on next page)

(continued from previous page)

```
    print(key, value)

print("-"*10)
# NOTE: Checking if a key exists
key = 'e'
if key in adict: # NO .keys() here please!
    print(adict[key])
else:
    print("Not found!")
```

```
{'a': 1, 'b': 2, 'c': 3}
dict_keys(['a', 'b', 'c'])
1
----------
a 1
b 2
c 3
e 5
----------
a 1
b 2
c 3
e 5
----------
5
```

```
"""
Special dictionaries
"""
# set is a dictionary without values
aset = set()
aset.add('a')

# deduplication short-cut using set
alist = [1,2,3,3,3,4,3]
alist = list(set(alist))
print(alist)

# default_dictionary returns a value computed from a default function
#     for non-existent entries
from collections import defaultdict
adict = defaultdict(lambda: 'unknown')
adict['cat'] = 'feline'
print(adict['cat'])
print(adict['dog'])
```

```
[1, 2, 3, 4]
feline
unknown
```

```
# counter is a dictionary with default value of 0
#     and provides handy iterable counting tools
from collections import Counter

# initialize and modify empty counter
```

(continues on next page)

(continued from previous page)

```
counter1 = Counter()
counter1['t'] = 10
counter1['t'] += 1
counter1['e'] += 1
print(counter1)
print("-"*10)

# initialize counter from iterable
counter2 = Counter("letters to be counted")
print(counter2)
print("-"*10)

# computations using counters
print("1", counter1 + counter2)
print("2,", counter1 - counter2)
print("3", counter1 or counter2) # or for intersection, and for union
```

```
Counter({'t': 11, 'e': 1})
----------
Counter({'e': 4, 't': 4, ' ': 3, 'o': 2, 'l': 1, 'r': 1, 's': 1, 'b': 1, 'c': 1, 'u':␣
↪1, 'n': 1, 'd': 1})
----------
1 Counter({'t': 15, 'e': 5, ' ': 3, 'o': 2, 'l': 1, 'r': 1, 's': 1, 'b': 1, 'c': 1, 'u
↪': 1, 'n': 1, 'd': 1})
2, Counter({'t': 7})
3 Counter({'t': 11, 'e': 1})
```

```
# sorting
a = [4,6,1,7,0,5,1,8,9]
a = sorted(a)
print(a)
a = sorted(a, reverse=True)
print(a)
```

```
[0, 1, 1, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 1, 1, 0]
```

```
# sorting
a = [("cat",1), ("dog", 3), ("bird", 2)]
a = sorted(a)
print(a)
a = sorted(a, key=lambda x:x[1])
print(a)
```

```
[('bird', 2), ('cat', 1), ('dog', 3)]
[('cat', 1), ('bird', 2), ('dog', 3)]
```

```
# useful in dictionary sorting
adict = {'cat':3, 'bird':1}
print(sorted(adict.items(), key=lambda x:x[1]))
```

```
[('bird', 1), ('cat', 3)]
```

```
# Syntax sugar: one-line control flow + list operation
sent = ["i am good", "a beautiful day", "HELLO FRIEND"]
"""
for i in range(len(sent)):
    sent[i] = sent[i].lower().split(" ")
"""
sent1 = [s.lower().split(" ") for s in sent]
print(sent1)

sent2 = [s.lower().split(" ") for s in sent if len(s) > 10]
print(sent2)

# Use this for deep copy!
# copy = [obj.copy() for obj in original]
```

```
[['i', 'am', 'good'], ['a', 'beautiful', 'day'], ['hello', 'friend']]
[['a', 'beautiful', 'day'], ['hello', 'friend']]
```

```
# Syntax sugar: * operator for repeating iterable elements
print("-"*10)
print([1]*10)

# Note: This only repeating by value
#       So you cannot apply the trick on reference types

# To create a double list
# DONT
doublelist = [[]]*10
doublelist[0].append(1)
print(doublelist)
# DO
doublelist = [[] for _ in range(10)]
doublelist[0].append(1)
print(doublelist)
```

```
----------
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
[[1], [], [], [], [], [], [], [], [], []]
```

1.10 Numpy

Very powerful python tool for handling matrices and higher dimensional arrays

```
import numpy as np
```

```
# create arrays
a = np.array([[1,2],[3,4],[5,6]])
print(a)
print(a.shape)
# create all-zero/one arrays
b = np.ones((3,4)) # np.zeros((3,4))
print(b)
```

(continues on next page)

(continued from previous page)

```
print(b.shape)
# create identity matrix
c = np.eye(5)
print(c)
print(c.shape)
```

```
[[1 2]
 [3 4]
 [5 6]]
(3, 2)
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
(3, 4)
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
(5, 5)
```

```
# reshaping arrays
a = np.arange(8)          # [8,] similar range() you use in for-loops
b = a.reshape((4,2))      # shape [4,2]
c = a.reshape((2,2,-1))   # shape [2,2,2] -- -1 for auto-fill
d = c.flatten()           # shape [8,]
e = np.expand_dims(a, 0) # [1,8]
f = np.expand_dims(a, 1) # [8,1]
g = e.squeeze()           # shape[8, ]    -- remove all unnecessary dimensions
print(a)
print(b)
```

```
[0 1 2 3 4 5 6 7]
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

```
# concatenating arrays
a = np.ones((4,3))
b = np.ones((4,3))
c = np.concatenate([a,b], 0)
print(c.shape)
d = np.concatenate([a,b], 1)
print(d.shape)
```

```
(8, 3)
(4, 6)
```

```
# one application is to create a batch for NN
x1 = np.ones((32,32,3))
x2 = np.ones((32,32,3))
x3 = np.ones((32,32,3))
# --> to create a batch of shape (3,32,32,3)
```

(continues on next page)

(continued from previous page)

```
x = [x1, x2, x3]
x = [np.expand_dims(xx, 0) for xx in x] # xx shape becomes (1,32,32,3)
x = np.concatenate(x, 0)
print(x.shape)
```

```
(3, 32, 32, 3)
```

```
# access array slices by index
a = np.zeros([10, 10])
a[:3] = 1
a[:, :3] = 2
a[:3, :3] = 3
rows = [4,6,7]
cols = [9,3,5]
a[rows, cols] = 4
print(a)
```

```
[[3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 4.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 4. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 4. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]]
```

```
# transposition
a = np.arange(24).reshape(2,3,4)
print(a.shape)
print(a)
a = np.transpose(a, (2,1,0)) # swap 0th and 2nd axes
print(a.shape)
print(a)
```

```
(2, 3, 4)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
(4, 3, 2)
[[[ 0 12]
  [ 4 16]
  [ 8 20]]

 [[ 1 13]
  [ 5 17]
  [ 9 21]]

 [[ 2 14]
  [ 6 18]
```

(continues on next page)

(continued from previous page)

```
  [10 22]]

 [[ 3 15]
  [ 7 19]
  [11 23]]]
```

```
c = np.array([[1,2],[3,4]])
# pinv is pseudo inversion for stability
print(np.linalg.pinv(c))
# l2 norm by default, read documentation for more options
print(np.linalg.norm(c))
# summing a matrix
print(np.sum(c))
# the optional axis parameter
print(c)
print(np.sum(c, axis=0)) # sum along axis 0
print(np.sum(c, axis=1)) # sum along axis 1
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
5.477225575051661
10
[[1 2]
 [3 4]]
[4 6]
[3 7]
```

```
# dot product
c = np.array([1,2])
d = np.array([3,4])
print(np.dot(c,d))
```

```
11
```

```
# matrix multiplication
a = np.ones((4,3)) # 4,3
b = np.ones((3,2)) # 3,2 --> 4,2
print(a @ b)       # same as a.dot(b)
c = a @ b          # (4,2)

# automatic repetition along axis
d = np.array([1,2,3,4]).reshape(4,1)
print(c + d)
# handy for batch operation
batch = np.ones((3,32))
weight = np.ones((32,10))
bias = np.ones((1,10))
print((batch @ weight + bias).shape)
```

```
[[3. 3.]
 [3. 3.]
 [3. 3.]
 [3. 3.]]
[[4. 4.]
```

(continues on next page)

(continued from previous page)

```
 [5. 5.]
 [6. 6.]
 [7. 7.]]
(3, 10)
```

```
# speed test: numpy vs list
a = np.ones((100,100))
b = np.ones((100,100))

def matrix_multiplication(X, Y):
    result = [[0]*len(Y[0]) for _ in range(len(X))]
    for i in range(len(X)):
        for j in range(len(Y[0])):
            for k in range(len(Y)):
                result[i][j] += X[i][k] * Y[k][j]
    return result

import time

# run numpy matrix multiplication for 10 times
start = time.time()
for _ in range(10):
    a @ b
end = time.time()
print("numpy spends {} seconds".format(end-start))

# run list matrix multiplication for 10 times
start = time.time()
for _ in range(10):
    matrix_multiplication(a,b)
end = time.time()
print("list operation spends {} seconds".format(end-start))

# the difference gets more significant as matrices grow in size!
```

```
numpy spends 0.001990079879760742 seconds
list operation spends 8.681961059570312 seconds
```

```
# element-wise operations, for examples
np.log(a)
np.exp(a)
np.sin(a)
# operation with scalar is interpreted as element-wise
a * 3
```

```
array([[3., 3., 3., ..., 3., 3., 3.],
       [3., 3., 3., ..., 3., 3., 3.],
       [3., 3., 3., ..., 3., 3., 3.],
       ...,
       [3., 3., 3., ..., 3., 3., 3.],
       [3., 3., 3., ..., 3., 3., 3.],
       [3., 3., 3., ..., 3., 3., 3.]])
```

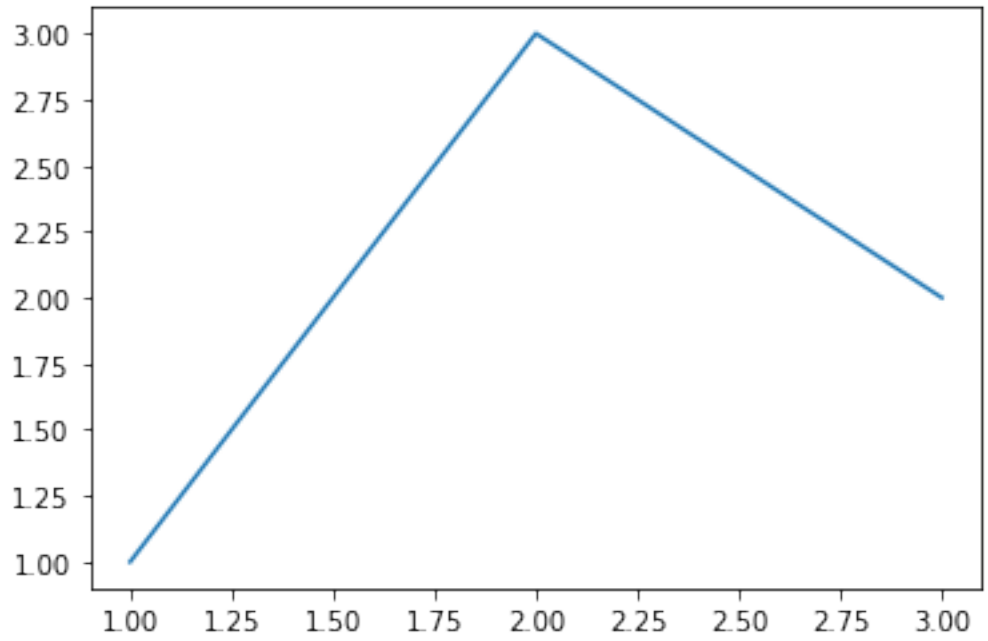
1.11 Matplotlib

Powerful tool for visualization Many tutorials online. We only go over the basics here

```
import matplotlib.pyplot as plt
```

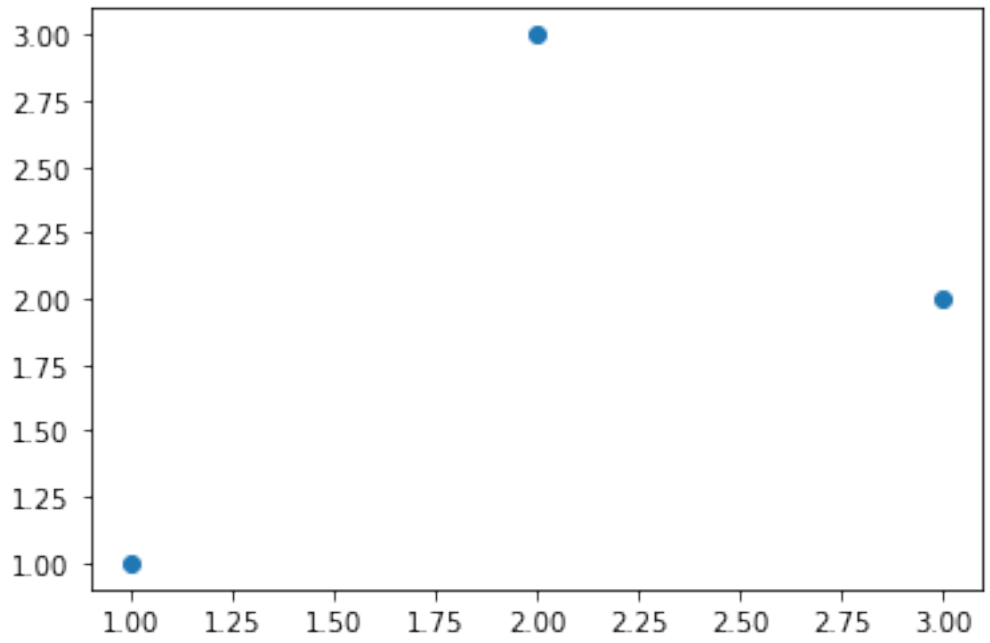
```
# line plot
x = [1,2,3]
y = [1,3,2]
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x17056d253a0>]
```



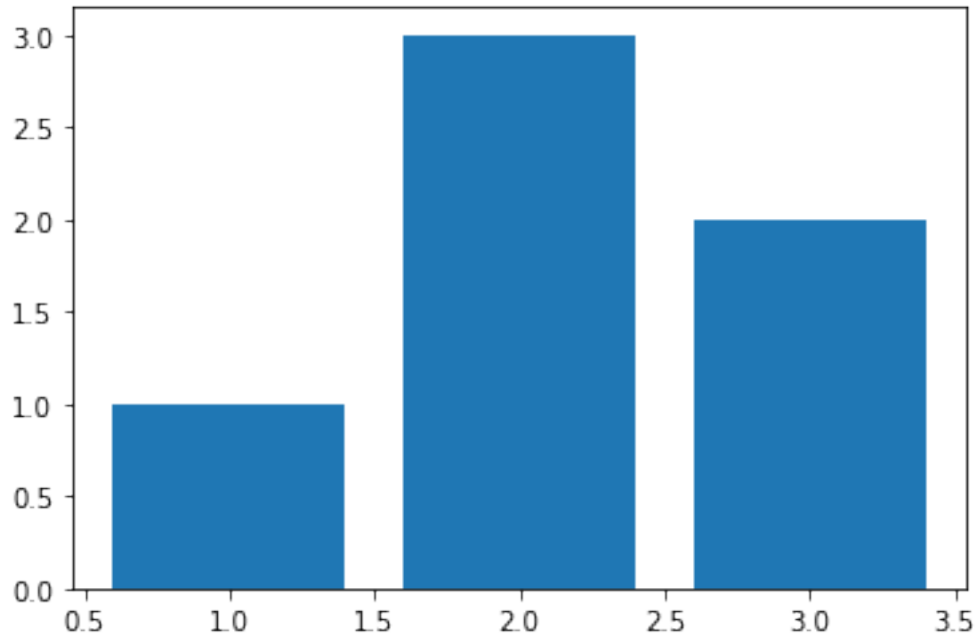

```
# scatter plot
plt.scatter(x,y)
```

```
<matplotlib.collections.PathCollection at 0x17b1b530490>
```




```
# bar plots
plt.bar(x,y)
```

```
<BarContainer object of 3 artists>
```



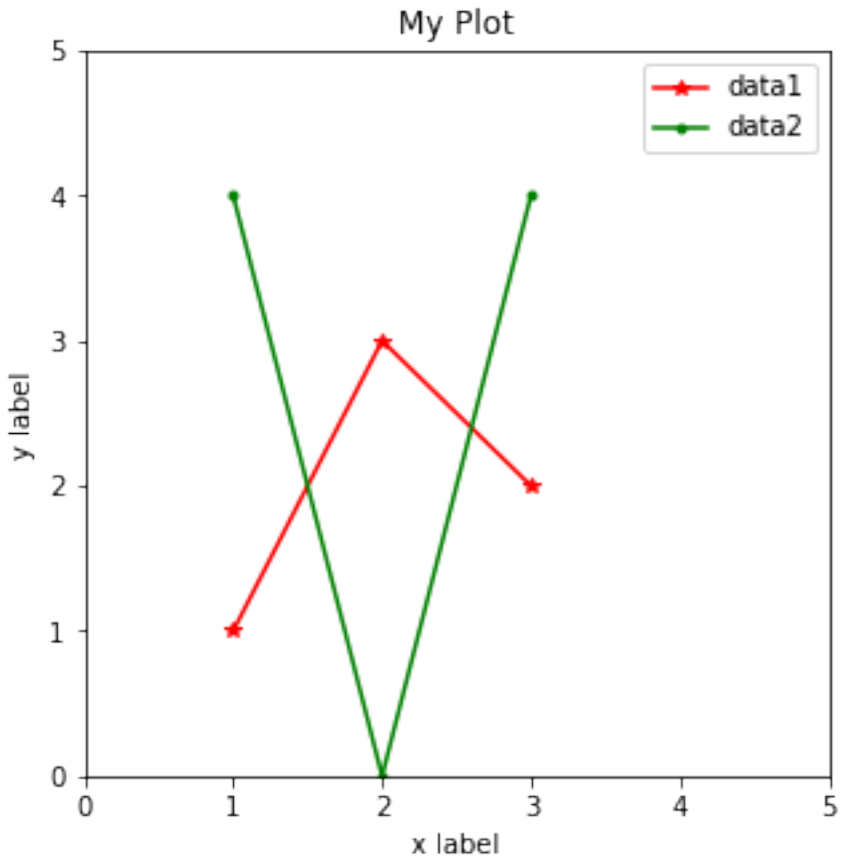

```
# plot configurations
x = [1,2,3]
y1 = [1,3,2]
y2 = [4,0,4]
```

(continues on next page)

(continued from previous page)

```
# set figure size
plt.figure(figsize=(5,5))

# set axes
plt.xlim(0,5)
plt.ylim(0,5)
plt.xlabel("x label")
plt.ylabel("y label")

# add title
plt.title("My Plot")

plt.plot(x,y1, label="data1", color="red", marker="*")
plt.plot(x,y2, label="data2", color="green", marker=".")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x17b1b669d00>
```



CHAPTER

TWO

LAB02: NLTK

It is easy to get our hands on millions of words of text. What can we do with it, assuming we can write some simple programs? In this chapter we'll address the following questions:

- What can we achieve by combining simple programming techniques with large quantities of text?
- How can we automatically extract key words and phrases that sum up the style and content of a text?
- What tools and techniques does the Python programming language provide for such work?
- What are some of the interesting challenges of natural language processing?

This chapter is divided into sections that skip between two quite different styles.

- In the "computing with language" sections we will take on some linguistically motivated programming tasks without necessarily explaining how they work.
- In the "closer look at Python" sections we will systematically review key programming concepts.

You can skip Section 2 and 4 if you are confident with Python, but do attempt the exercises and refer back to those sections if you find some of the questions challenging.

We'll flag the two styles in the section titles, but later labs will mix both styles without being so up-front about it. We hope this style of introduction gives you an authentic taste of what will come later, while covering a range of elementary concepts in linguistics and computer science.

Online reference material for NLTK is at http://nltk.org/.

This lab may raise more questions than it answers, we will try to address some in the rest of this unit.

Original Content Credit to Chapter 1 of the online NLTK Book

2.1 Starting with NLTK

If you have installed nltk and have downloaded the data and models, you can skip this.

```
import nltk
nltk.download()
```

Downloading the NLTK Book Collection: browse the available packages using nltk.download(). The Collections tab on the downloader shows how the packages are grouped into sets, and you should select the line labeled book to obtain all data required for the examples and exercises in this book. It consists of about 30 compressed files requiring about 100Mb disk space. The full collection of data (i.e., all in the downloader) is nearly ten times this size (at the time of writing) and continues to expand.

Once the data is downloaded to your machine, you can load some of it using the Jupyter Notebook. The first step is to type a special command which tells the interpreter to load some texts for us to explore: `from nltk.book import`

*. This says "from NLTK's book module, load all items." The book module contains all the data you will need as you read this chapter. After printing a welcome message, it loads the text of several books (this will take a few seconds). Here's the command again, together with the output that you will see. Take care to get spelling and punctuation right.

```
from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

Any time we want to find out about these texts, we just have to enter their names at the Python prompt:

```
text1
```

```
<Text: Moby Dick by Herman Melville 1851>
```

```
text2
```

```
<Text: Sense and Sensibility by Jane Austen 1811>
```

Now that we have some data to work with, we're ready to get started.

2.1.1 Searching Text

Concordance

There are many ways to examine the context of a text apart from simply reading it. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word `monstrous` in Moby Dick:

```
text1.concordance("monstrous")
```

```
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the Monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
```

The first time you use a concordance on a particular text, it takes a few extra seconds to build an index so that subsequent searches are fast.

Your Turn

- Try searching for other words,
- You can also try searches on some of the other texts we have included. For example, search `Sense and Sensibility` for the word `affection`, using `text2`.concordance("affection").
- Search the book of `Genesis` to find out how long some people lived, using `text3.concordance("lived")`.
- You could look at `text4`, `the Inaugural Address Corpus`, to see examples of English going back to 1789, and search for words like `nation`, `terror`, `god` to see how these words have been used differently over time.
- We've also included `text5`, the `NPS Chat Corpus`: search this for unconventional words like `im`, `ur`, `lol`. (Note that this corpus is uncensored!)

Once you've spent a little while examining these texts, we hope you have a new sense of the richness and diversity of language.

Similar Words and Common Context

A concordance permits us to see words in context. For example, we saw that monstrous occurred in contexts such as the ___ pictures and a ___ size . What other words appear in a similar range of contexts? We can find out by using `similar()` function for the text object (e.g. `text1`) with the word (e.g. `monstrous`) as its argument:

```
text1.similar("monstrous")
```

```
true contemptible christian abundant few part mean careful puzzled
mystifying passing curious loving wise doleful gamesome singular
delightfully perilous fearless
```

```
text2.similar("monstrous")
```

```
very so exceedingly heartily a as good great extremely remarkably
sweet vast amazingly
```

Observe that we get different results for different texts. Austen uses this word quite differently from Melville; for her, `monstrous` has positive connotations, and sometimes functions as an intensifier like the word `very`.

The term common_contexts allows us to examine just the contexts that are shared by two or more words, such as monstrous and very. We have to enclose these words by square brackets as well as parentheses, and separate them with a comma:

```
text2.common_contexts(["monstrous", "very"])
```

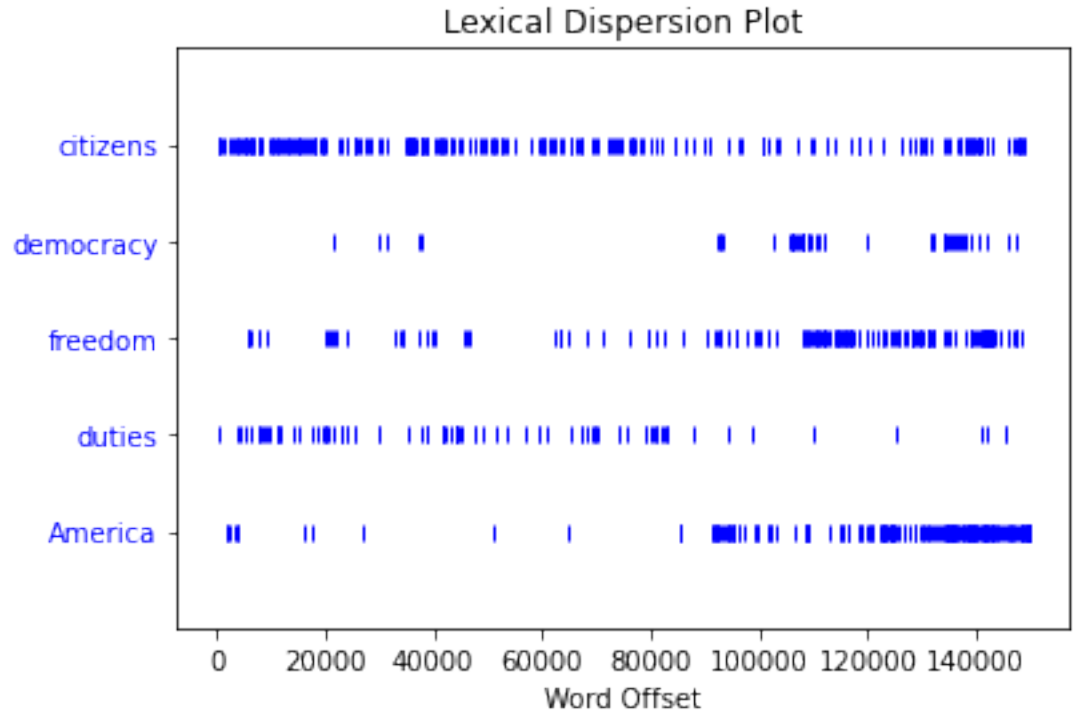
```
am_glad a_pretty a_lucky is_pretty be_glad
```

Your Turn

Pick another pair of words and compare their usage in two different texts, using the `similar()` and `common_contexts()` functions.

We have seen how to automatically detect that a particular word occurs in a text, and to display some words that appear in the same context. We can also determine the location of a word in the text: how many words from the beginning it appears. This positional information can be displayed using a dispersion plot. Each stripe represents an instance of a word, and each row represents the entire text. We can see the dispersion of words in `text4` (Inaugural Address Corpus). You can produce this plot as shown below. You might like to try more words (e.g., `liberty`, `constitution`), and different texts. Can you predict the dispersion of a word before you view it? As before, take care to get the quotes, commas, brackets and parentheses exactly right.

```
text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
```



Important: You need to have Python's NumPy and Matplotlib packages installed in order to produce the graphical plots used in this book. Please see http://nltk.org/ for installation instructions.

You can also plot the frequency of word usage through time using https://books.google.com/ngrams

Generating Text

Now, just for fun, let's try generating some random text in the various styles we have just seen. To do this, we type the name of the text followed by the term generate. (We need to include the parentheses, but there's nothing that goes between them.)

```
text3.generate()
```

```
Building ngram index...
```

```
laid by her , and said unto Cain , Where art thou , and said , Go to ,
I will not do it for ten ' s sons ; we dreamed each man according to
```

(continues on next page)

(continued from previous page)

```
their generatio the firstborn said unto Laban , Because I said , Nay ,
but Sarah shall her name be . , duke Elah , duke Shobal , and Akan .
and looked upon my affliction . Bashemath Ishmael ' s blood , but Isra
for as a prince hast thou found of all the cattle in the valley , and
the wo The
```

```
"laid by her , and said unto Cain , Where art thou , and said , Go to ,\nI will not␣
↪do it for ten ' s sons ; we dreamed each man according to\ntheir generatio the␣
↪firstborn said unto Laban , Because I said , Nay ,\nbut Sarah shall her name be . ,␣
↪duke Elah , duke Shobal , and Akan .\nand looked upon my affliction . Bashemath␣
↪Ishmael ' s blood , but Isra\nfor as a prince hast thou found of all the cattle in␣
↪the valley , and\nthe wo The"
```

2.1.2 Counting Vocabulary

The most obvious fact about texts that emerges from the preceding examples is that they differ in the vocabulary they use. In this section we will see how to use the computer to count the words in a text in a variety of useful ways. Test your understanding by modifying the examples, and trying the exercises at the end of this lab.

Text Size vs. Vocabulary Size

Let's begin by finding out the length of a text from start to finish, in terms of the words and punctuation symbols that appear. We use the term len to get the length of something, which we'll apply here to the book of `Genesis`:

```
len(text3)
```

```
44764
```

So `Genesis` has 44,764 words and punctuation symbols, or "tokens." A token is the technical name for a sequence of characters — such as `hairy`, `his`, or `:)` — that we want to treat as a group. When we count the number of tokens in a text, say, the phrase `to be or not to be`, we are counting occurrences of these sequences. Thus, in our example phrase there are two occurrences of `to`, two of `be`, and one each of `or` and `not`. But there are only four distinct vocabulary items in this phrase.

Your Turn

How many distinct words does the book of `Genesis` contain?

To work this out in Python, we have to pose the question slightly differently. The vocabulary of a text is just the set of tokens that it uses, since in a set, all duplicates are collapsed together. In Python we can obtain the vocabulary items of `text3` with the command: `set(text3)`. When you do this, many screens of words will fly past. Now try the following. By wrapping sorted() around the Python expression `set(text3)`, we obtain a sorted list of vocabulary items, beginning with various punctuation symbols and continuing with words starting with A. All capitalized words precede lowercase words. `[:20]` will list the first 20 tokens, not including the token indexed by `20`.

```
sorted(set(text3)) [:20]
```

```
['!',
 "'",
 '(',
```

(continues on next page)

(continued from previous page)

```
 ')',
 ',',
 ',)',
 '.',
 '.)',
 ':',
 ';',
 ';)',
 '?',
 '?)',
 'A',
 'Abel',
 'Abelmizraim',
 'Abidah',
 'Abide',
 'Abimael',
 'Abimelech']
```

```
len(set(text3))
```

```
2789
```

We discover the size of the vocabulary indirectly, by asking for the number of items in the set, and again we can use len to obtain this number. Although it has 44,764 tokens, this book has only 2,789 distinct words, or "word types." A word type is the form or spelling of the word independently of its specific occurrences in a text — that is, the word considered as a unique item of vocabulary. Our count of 2,789 items will include punctuation symbols, so we will generally call these unique items types instead of word types.

Lexical Richness

Now, let's calculate a measure of the lexical richness of the text. The next example shows us that the number of distinct words is just 6% of the total number of words, or equivalently that each word is used 16 times on average.

```
len(set(text3)) / len(text3)
```

```
0.06230453042623537
```

Word Freqency

Next, let's focus on particular words. We can count how often a word occurs in a text, and compute what percentage of the text is taken up by a specific word:

```
# raw count
text3.count("smote")
```

```
5
```

```
# percentage frequency
100 * text4.count('a') / len(text4)
```

```
1.457973123627309
```

Your Turn

How many times does the word `lol` appear in `text5`? How much is this as a percentage of the total number of words in this text?

Your Turn

You may want to repeat such calculations on several texts, but it is tedious to keep retyping the formula. Instead, you can come up with your own name for a task, like "lexical_diversity" or "tf" (for term frequency), and associate it with a block of code. Now you only have to type a short name instead of one or more complete lines of Python code, and you can re-use it as often as you like. The block of code that does a task for us is called a **function**, and we define a short name for our function with the keyword def. The next example shows how to define two new functions, `lexical_diversity()` and `tf()`:

```
def lexical_diversity(text):
    return len(set(text)) / len(text)
```

```
def tf(text, token):
    count = text.count(token)
    total = len(text)
    return 100 * count / total
```

In the definition of `lexical_diversity()`, we specify a parameter named text . This parameter is a "placeholder" for the actual text whose lexical diversity we want to compute, and reoccurs in the block of code that will run when the function is used. Similarly, `tf()` is defined to take two parameters, named `text` and `token`.

Once Python knows that `lexical_diversity()` and `tf()` are the names for specific blocks of code, we can go ahead and use these functions:

```
lexical_diversity(text3)
```

```
0.06230453042623537
```

```
tf(text4, 'a')
```

```
1.457973123627309
```

To recap, we use or call a function such as `lexical_diversity()` by typing its name, followed by an open parenthesis, the name of the text, and then a close parenthesis. These parentheses will show up often; their role is to separate the name of a task — such as `lexical_diversity()` — from the data that the task is to be performed on — such as `text3`. The data value that we place in the parentheses when we call a function is an argument to the function.

You have already encountered several functions in this lab, such as `len()`, `set()`, and sorted(). By convention, we will always add an empty pair of parentheses after a function name, as in `len()`, just to make clear that what we are talking about is a function rather than some other kind of Python expression. Functions are an important concept in programming, you should consider refactoring frequently reusable blocks of code into functions.

Later we'll see how to use functions when tabulating data. Each row of the table will involve the same computation but with different data, and we'll do this repetitive work using a function.

Table 2.1: Lexical Diversity of Various Genres in the Brown Corpus

Genre	Tokens	Types	Lexical diversity
skill and hobbies	82345	11935	0.145
humor	21695	5017	0.231
fiction: science	14470	3233	0.223
press: reportage	100554	14394	0.143
fiction: romance	70022	8452	0.121
religion	39399	6373	0.162

2.2 A Closer Look at Python: Texts as Lists of Words

2.2.1 Lists

What is a text? At one level, it is a sequence of symbols on a page such as this one. At another level, it is a sequence of chapters, made up of a sequence of sections, where each section is a sequence of paragraphs, and so on. However, for our purposes, we will think of a text as nothing more than a sequence of words and punctuation. Here's how we represent text in Python, in this case the opening sentence of `Moby Dick`:

```
sent1 = ['Call', 'me', 'Ishmael', '.']
```

Here we have given a variable name we made up, `sent1`, followed by the equals sign, and then some quoted words, separated with commas, and surrounded with a pair of square brackets. This square bracketed, comma seperated content is known as a *list* in Python: it is how we store a text. We can inspect it, ask for its length and apply our own lexical_diversity() function to it.

```
sent1
```

```
['Call', 'me', 'Ishmael', '.']
```

```
len(sent1)
```

```
4
```

To use the functions we defined in another Jupyter Notebook, the best practice is to create a plain python (.py) to host all the functions, rather than import the ipython file through packages like nbimporter.

After copying the two functions into a file called `utils.py`, we can import the functions and use them in the current notebook.

```
from utils import lexical_diversity
lexical_diversity(sent1)
```

```
1.0
```

Some more lists have been defined for you, one for the opening sentence of each of our texts, sent2 … sent9. We inspect two of them here.

```
from nltk.book import *
sent2
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

```
['The',
 'family',
 'of',
 'Dashwood',
 'had',
 'long',
 'been',
 'settled',
 'in',
 'Sussex',
 '.']
```

```
sent3
```

```
['In',
 'the',
 'beginning',
 'God',
 'created',
 'the',
 'heaven',
 'and',
 'the',
 'earth',
 '.']
```

Your Turn

Make up a few sentences of your own, by typing a name, equals sign, and a list of words, like this: `ex1 = ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']`. Repeat some of the other Python operations we saw earlier in 1, e.g., `sorted(ex1)`, `len(set(ex1))`, `ex1.count('the')`.

A pleasant surprise is that we can use Python's addition operator on lists. Adding two lists creates a new list with everything from the first list, followed by everything from the second list:

```
['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail']
```

```
['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

This special use of the addition operation is called concatenation; it combines the lists together into a single list. We can concatenate sentences to build up a text.

We don't have to literally type the lists either; we can use short names that refer to pre-defined lists.

```
sent4 + sent1
```

```
['Fellow',
 '-',
 'Citizens',
 'of',
 'the',
 'Senate',
 'and',
 'of',
 'the',
 'House',
 'of',
 'Representatives',
 ':',
 'Call',
 'me',
 'Ishmael',
 '.']
```

What if we want to add a single item to a list? This is known as appending. When we `append()` to a list, the list itself is updated as a result of the operation.

```
sent1.append("Some")
sent1
```

```
['Call', 'me', 'Ishmael', '.', 'Some']
```

2.2.2 Indexing Lists

As we have seen, a text in Python is a list of words, represented using a combination of brackets and quotes. Just as with an ordinary page of text, we can count up the total number of words in text1 with `len(text1)`, and count the occurrences in a text of a particular word — say, `'heaven'` — using `text1.count('heaven')`.

With some patience, we can pick out the 1st, 173rd, or even 14,278th word in a printed text. Analogously, we can identify the elements of a Python list by their order of occurrence in the list. The number that represents this position is the item's index. We instruct Python to show us the item that occurs at an index such as 173 in a text by writing the name of the text followed by the index inside square brackets:

```
text4[173]
```

```
'awaken'
```

We can do the converse; given a word, find the index of *when it first occurs*:

```
text4.index('awaken')
```

```
173
```

Indexes are a common way to access the words of a text, or, more generally, the elements of any list. Python permits us to access sublists as well, extracting manageable pieces of language from large texts, a technique known as *slicing*.

```
text5[16715:16735]
```

```
['U86',
 'thats',
 'why',
 'something',
 'like',
 'gamefly',
 'is',
 'so',
 'good',
 'because',
 'you',
 'can',
 'actually',
 'play',
 'a',
 'full',
 'game',
 'without',
 'buying',
 'it']
```

Indexes have some subtleties, and we'll explore these with the help of an artificial sentence:

```
sent = ['word1', 'word2', 'word3', 'word4', 'word5',
...         'word6', 'word7', 'word8', 'word9', 'word10']
sent[0]
```

```
'word1'
```

```
sent[9]
```

```
'word10'
```

Caution: Notice that our indexes start from zero: sent element zero, written sent[0], is the first word, 'word1', whereas sent element 9 is 'word10'. The reason is simple: the moment Python accesses the content of a list from the computer's memory, it is already at the first element; we have to tell it how many elements forward to go. Thus, zero steps forward leaves it at the first element.

This practice of counting from zero is initially confusing, but typical of modern programming languages. You'll quickly get the hang of it if you've mastered the system of counting centuries where 19XY is a year in the 20th century, or if you live in a country where the floors of a building are numbered from 1, and so walking up n-1 flights of stairs takes you to level n.

if we accidentally use an index that is too large, we get an error:

```
sent[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-24-baa8b4ccd19d> in <module>
----> 1 sent[10]
```

(continues on next page)

(continued from previous page)

```
IndexError: list index out of range
```

This time it is *not a syntax error*, because the program fragment is syntactically correct. Instead, it is *a runtime error*, and it produces a Traceback message that shows the context of the error, followed by the name of the error, IndexError, and a brief explanation.

Let's take a closer look at slicing, using our artificial sentence again. Here we verify that the slice 5:8 includes sent elements at indexes 5, 6, and 7:

```
sent[5:8]
```

```
['word6', 'word7', 'word8']
```

By convention, m:n means elements m…n-1 inclusive. As the next example shows, we can omit the first number if the slice begins at the start of the list, and we can omit the second number if the slice goes to the end:

```
sent[:3]
```

```
['word1', 'word2', 'word3']
```

```
text2[141525:]
```

```
['among',
 'the',
 'merits',
 'and',
 'the',
 'happiness',
 'of',
 'Elinor',
 'and',
 'Marianne',
 ',',
 'let',
 'it',
 'not',
 'be',
 'ranked',
 'as',
 'the',
 'least',
 'considerable',
 ',',
 'that',
 'though',
 'sisters',
 ',',
 'and',
 'living',
 'almost',
 'within',
 'sight',
 'of',
 'each',
```

(continues on next page)

(continued from previous page)

```
 'other',
 ',',
 'they',
 'could',
 'live',
 'without',
 'disagreement',
 'between',
 'themselves',
 ',',
 'or',
 'producing',
 'coolness',
 'between',
 'their',
 'husbands',
 '.',
 'THE',
 'END']
```

We can modify an element of a list by assigning to one of its index values. In the next example, we put `sent[0]` on the left of the equals sign. We can also replace an entire slice with new elements. A consequence of this last change is that the list only has four elements, and accessing a later value generates an error.

```
sent[0] = 'First'
sent[9] = 'Last'
len(sent)
```

```
10
```

```
sent[1:9] = ['Second', 'Third']
sent
```

```
['First', 'Second', 'Third', 'Last']
```

```
sent[9]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-31-9d73df3f8677> in <module>
----> 1 sent[9]

IndexError: list index out of range
```

Your Turn

Take a few minutes to define a sentence of your own and modify individual words and groups of words (slices) using the same methods used earlier. Check your understanding by trying the exercises on lists at the end of this chapter.

2.3 Variables

The basic form of Python statements is: `variable = expression`. Python will evaluate the expression, and save its result to the variable. This process is called assignment. It does not generate any output; you have to type the variable on a line of its own to inspect its contents. The equals sign is slightly misleading, since information is moving from the

right side to the left. It might help to think of it as a left-arrow. The name of the variable can be anything you like, e.g., my_sent, sentence, xyzzy. It must start with a letter, and can include numbers and underscores. Here are some examples of variables and assignments:

Note: Remember that capitalized words appear before lowercase words in sorted lists.

Notice Python expressions can be split across multiple lines, so long as this happens within any kind of brackets. It doesn't matter how much indentation is used in these continuation lines, but some indentation usually makes them easier to read.

It is good to choose meaningful variable names to remind you — and to help anyone else who reads your Python code — what your code is meant to do. Python does not try to make sense of the names; it blindly follows your instructions, and does not object if you do something confusing, such as `one = 'two'` or `two = 3`. The only restriction is that a variable name cannot be any of Python's reserved words, such as `def`, `if`, `not`, and `import`. If you use a reserved word, Python will produce a syntax error:

```
not = 'Camelot'
```

```
  File "<ipython-input-32-4961bbbbef94>", line 1
    not = 'Camelot'
        ^
SyntaxError: invalid syntax
```

We will often use variables to hold intermediate steps of a computation, especially when this makes the code easier to follow. Thus `len(set(text1))` could also be written:

```
vocab = set(text1)
vocab_size = len(vocab)
vocab_size
```

```
19317
```

Caution: Take care with your choice of names (or identifiers) for Python variables. First, you should start the name with a letter, optionally followed by digits (0 to 9) or letters. Thus, abc23 is fine, but 23abc will cause a syntax error. Names are case-sensitive, which means that myVar and myvar are distinct variables. Variable names cannot contain whitespace, but you can separate words using an underscore, e.g., my_var. Be careful not to insert a hyphen instead of an underscore: my-var is wrong, since Python interprets the "-" as a minus sign.

2.2.3 Strings

Some of the methods we used to access the elements of a list also work with individual words, or strings. For example, we can assign a string to a variable [1], index a string, and slice a string:

```
name = 'Monty'
name[0]
```

```
'M'
```

```
name[:4]
```

```
'Mont'
```

We can also perform multiplication and addition with strings:

```
name * 2
```

```
'MontyMonty'
```

```
name + '!'
```

```
'Monty!'
```

We can join the words of a list to make a single string, or split a string into a list, as follows:

```
' '.join(['Monty', 'Python'])
```

```
'Monty Python'
```

```
'Monty Python'.split()
```

We now have two important building blocks — lists and strings — and are ready to get back to some language analysis.

2.3 Computing with Language: Simple Statistics

Let us return to our exploration of the ways we can bring our computational resources to bear on large quantities of text. We began this discussion in 1, and saw how to search for words in context, how to compile the vocabulary of a text, how to generate random text in the same style, and so on.

In this section we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text.

Before continuing further, you might like to check your understanding of the last section by predicting the output of the following code. You can use the Jupyter Notebook Code Cell to check whether you got it right. If you're not sure how to do this task, it would be a good idea to review the previous section before continuing further.

```
saying = ['After', 'all', 'is', 'said', 'and', 'done',
          'more', 'is', 'said', 'than', 'done']
tokens = set(saying)
tokens = sorted(tokens)
tokens
```

```
['After', 'all', 'and', 'done', 'is', 'more', 'said', 'than']
```

Your Turn

What output do you expect of `tokens[-2:]?

2.3.1 Frequency Distributions

How can we automatically identify the words of a text that are most informative about the topic and genre of the text? Imagine how you might go about finding the 50 most frequent words of a book. One method would be to keep a tally for each vocabulary item. The tally would need thousands of rows (the same size as the vocabulary), and it would be an exceedingly laborious process — so laborious that we would rather assign the task to a machine. Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let's use a FreqDist to find the 50 most frequent words of `Moby Dick`:

```
from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

```
fdist1 = FreqDist(text1)
print(fdist1)
```

```
<FreqDist with 19317 samples and 260819 outcomes>
```

```
fdist1.most_common(50)
```

```
[(',', 18713),
 ('the', 13721),
 ('.', 6862),
 ('of', 6536),
 ('and', 6024),
 ('a', 4569),
 ('to', 4542),
 (';', 4072),
 ('in', 3916),
 ('that', 2982),
 ("'", 2684),
 ('-', 2552),
 ('his', 2459),
 ('it', 2209),
 ('I', 2124),
 ('s', 1739),
 ('is', 1695),
 ('he', 1661),
 ('with', 1659),
 ('was', 1632),
 ('as', 1620),
 ('"', 1478),
 ('all', 1462),
```

(continues on next page)

(continued from previous page)

```
 ('for', 1414),
 ('this', 1280),
 ('!', 1269),
 ('at', 1231),
 ('by', 1137),
 ('but', 1113),
 ('not', 1103),
 ('--', 1070),
 ('him', 1058),
 ('from', 1052),
 ('be', 1030),
 ('on', 1005),
 ('so', 918),
 ('whale', 906),
 ('one', 889),
 ('you', 841),
 ('had', 767),
 ('have', 760),
 ('there', 715),
 ('But', 705),
 ('or', 697),
 ('were', 680),
 ('now', 646),
 ('which', 640),
 ('?', 637),
 ('me', 627),
 ('like', 624)]
```

The tally is known as a frequency distribution, and it tells us the frequency of each vocabulary item in the text. (In general, it could count any kind of observable event.) It is a "distribution" because it tells us how the total number of word tokens in the text are distributed across the vocabulary items. Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let's use a FreqDist to find the 50 most frequent words of Moby Dick:

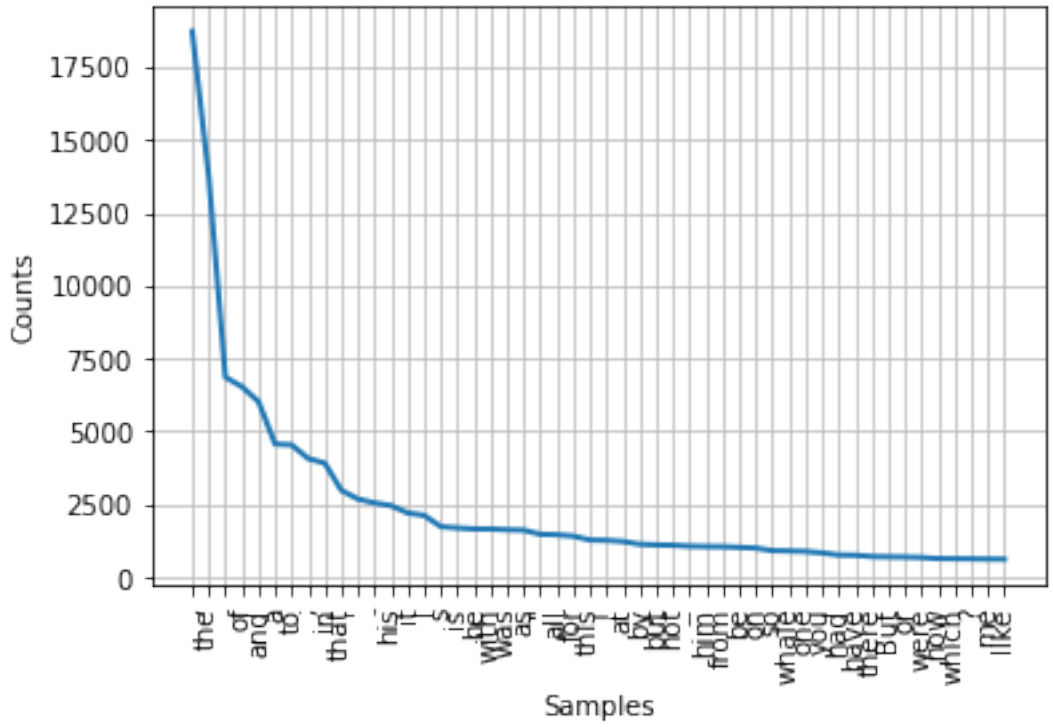
When we first invoke FreqDist, we pass the name of the text as an argument. We can inspect the total number of words ("outcomes") that have been counted up — 260,819 in the case of `Moby Dick`. The expression most_common(50) gives us a list of the 50 most frequently occurring types in the text with a raw count of each word.

Your Turn

Try the preceding frequency distribution example for yourself, for `text2`. Be careful to use the correct parentheses and uppercase letters. If you get an error message NameError: name 'FreqDist' is not defined, you need to start your work with from nltk.book import *

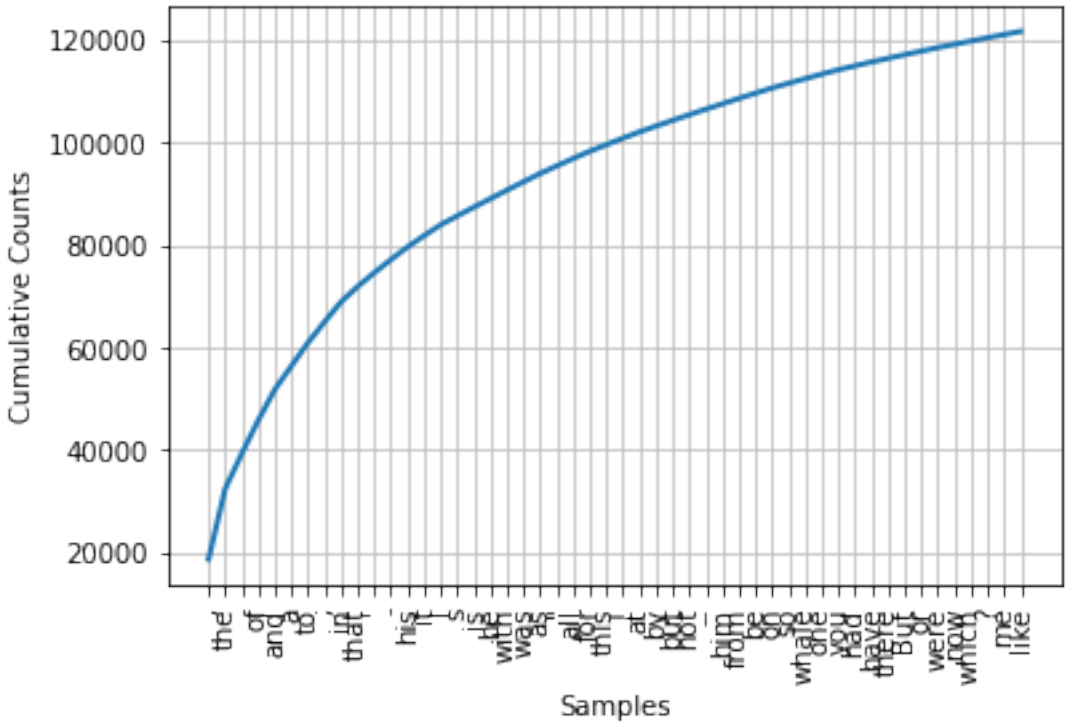
Do any words produced in the last example help us grasp the topic or genre of this text? Only one word, whale, is slightly informative! It occurs over 900 times. The rest of the words tell us nothing about the text; they're just English "plumbing." What proportion of the text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(50, cumulative=True)`, to produce the following graph. These 50 words account for nearly half the book!

```
# The frequency distribution
fdist1.plot(50)
```




```
<AxesSubplot:xlabel='Samples', ylabel='Counts'>
```

```
fdist1.plot(50, cumulative=True)
```



```
<AxesSubplot:xlabel='Samples', ylabel='Cumulative Counts'>
```

From the Cumulative Frequency Plot for 50 Most Frequently Words in `Moby Dick`: these account for nearly half of the tokens.

If the frequent words don't help us, how about the words that occur once only, the so-called hapaxes? View them by typing `fdist1.hapaxes()`. This list contains `lexicographer`, `cetological`, `contraband`, `expostulations`, and about 9,000 others. It seems that there are too many rare words, and without seeing the context we probably can't guess what half of the hapaxes mean in any case! Since neither frequent nor infrequent words help, we need to try something else.

```
fdist1.hapaxes()
```

```
['Herman',
 'Melville',
 ']',
 'ETYMOLOGY',
 'Late',
 'Consumptive',
 'School',
 'threadbare',
 'lexicons',
 'mockingly',
 'flags',
 'mortality',
 'signification',
 'HACKLUYT',
 'Sw',
 'HVAL',
 'roundness',
 'Dut',
 'Ger',
 'WALLEN',
 'WALW',
 'IAN',
 'RICHARDSON',
 'KETOS',
 'GREEK',
 'CETUS',
 'LATIN',
 'WHOEL',
 'ANGLO',
 'SAXON',
 'WAL',
 'HWAL',
 'SWEDISH',
 'ICELANDIC',
 'BALEINE',
 'BALLENA',
 'FEGEE',
 'ERROMANGOAN',
 'Librarian',
 'painstaking',
 'burrower',
 'grub',
 'Vaticans',
 'stalls',
```

(continues on next page)

(continued from previous page)

```
 'higgledy',
 'piggledy',
 'gospel',
 'promiscuously',
 'commentator',
 'belongest',
 'sallow',
 'Pale',
 'Sherry',
 'loves',
 'bluntly',
 'Subs',
 'thankless',
 'Hampton',
 'Court',
 'hie',
 'refugees',
 'pampered',
 'Michael',
 'Raphael',
 'unsplinterable',
 'GENESIS',
 'JOB',
 'JONAH',
 'punish',
 'ISAIAH',
 'soever',
 'cometh',
 'incontinently',
 'perisheth',
 'PLUTARCH',
 'MORALS',
 'breedeth',
 'Whirlpooles',
 'Balaene',
 'arpens',
 'PLINY',
 'Scarcely',
 'TOOKE',
 'LUCIAN',
 'TRUE',
 'catched',
 'OCTHER',
 'VERBAL',
 'TAKEN',
 'MOUTH',
 'ALFRED',
 '890',
 'gudgeon',
 'retires',
 'MONTAIGNE',
 'APOLOGY',
 'RAIMOND',
 'SEBOND',
 'Nick',
 'RABELAIS',
 'cartloads',
```

(continues on next page)

(continued from previous page)

```
 'STOWE',
 'ANNALS',
 'LORD',
 'BACON',
 'Touching',
 'ork',
 'DEATH',
 'sovereignest',
 'bruise',
 'HAMLET',
 'leach',
 'Mote',
 'availle',
 'returne',
 'againe',
 'worker',
 'Dinting',
 'paine',
 'thro',
 'maine',
 'FAERIE',
 'Immense',
 'til',
 'DAVENANT',
 'PREFACE',
 'GONDIBERT',
 'spermacetti',
 'Hosmannus',
 'Nescio',
 'VIDE',
 'Spencer',
 'Talus',
 'flail',
 'threatens',
 'jav',
 'lins',
 'WALLER',
 'SUMMER',
 'ISLANDS',
 'Commonwealth',
 'Civitas',
 'OPENING',
 'SENTENCE',
 'HOBBES',
 'LEVIATHAN',
 'Silly',
 'Mansoul',
 'chewing',
 'sprat',
 'PILGRIM',
 'PROGRESS',
 'Created',
 'PARADISE',
 'LOST',
 '---"',
 'Hugest',
 'Stretched',
```

(continues on next page)

(continued from previous page)

```
 'Draws',
 'FULLLER',
 'PROFANE',
 'HOLY',
 'STATE',
 'DRYDEN',
 'ANNUS',
 'MIRABILIS',
 'aground',
 'EDGE',
 'TEN',
 'SPITZBERGEN',
 'PURCHAS',
 'wantonness',
 'fuzzing',
 'vents',
 'HERBERT',
 'INTO',
 'ASIA',
 'AFRICA',
 'SCHOUTEN',
 'SIXTH',
 'CIRCUMNAVIGATION',
 'Elbe',
 'ducat',
 'herrings',
 'GREENLAND',
 'Several',
 'Fife',
 'Anno',
 '1652',
 'Pitferren',
 'SIBBALD',
 'FIFE',
 'KINROSS',
 'Myself',
 'Sperma',
 'ceti',
 'fierceness',
 'RICHARD',
 'STRAFFORD',
 'LETTER',
 'BERMUDAS',
 'PHIL',
 'TRANS',
 '1668',
 'PRIMER',
 'COWLEY',
 '1729',
 '"...',
 'frequendy',
 'insupportable',
 'disorder',
 'ULLOA',
 'SOUTH',
 'AMERICA',
 'sylphs',
```

(continues on next page)

(continued from previous page)

```
'petticoat',
'Oft',
'Tho',
'RAPE',
'LOCK',
'NAT',
'wales',
'JOHNSON',
'COOK',
'dung',
'lime',
'juniper',
'UNO',
'VON',
'TROIL',
'LETTERS',
'BANKS',
'SOLANDER',
'1772',
'Nantuckois',
'JEFFERSON',
'MEMORIAL',
'MINISTER',
'REFERENCE',
'PARLIAMENT',
'SOMEWHERE',
'guarding',
'protecting',
'robbers',
'BLACKSTONE',
'Rodmond',
'suspends',
'attends',
'FALCONER',
'Bright',
'roofs',
'domes',
'rockets',
'Around',
'unwieldy',
'COWPER',
'VISIT',
'LONDON',
'HUNTER',
'DISSECTION',
'SMALL',
'SIZED',
'aorta',
'gushing',
'PALEY',
'THEOLOGY',
'mammiferous',
'hind',
'BARON',
'CUVIER',
'COLNETT',
'PURPOSE',
```

(continues on next page)

(continued from previous page)

```
'EXTENDING',
'SPERMACETI',
'Floundered',
'chace',
'peopling',
'Gather',
'Led',
'instincts',
'trackless',
'Assaulted',
'voracious',
'spiral',
'MONTGOMERY',
'WORLD',
'FLOOD',
'Paean',
'fatter',
'Flounders',
'CHARLES',
'LAMB',
'TRIUMPH',
'1690',
'OBED',
'Susan',
'HAWTHORNE',
'TWICE',
'bespeak',
'raal',
'COOPER',
'PILOT',
'Berlin',
'Gazette',
'ECKERMANN',
'CONVERSATIONS',
'GOETHE',
'ESSEX',
'WAS',
'ATTACKED',
'FINALLY',
'DESTROYED',
'OWEN',
'CHACE',
'FIRST',
'SAID',
'VESSEL',
'YORK',
'1821',
'piping',
'dimmed',
'phospher',
'ELIZABETH',
'OAKES',
'SMITH',
'amounted',
'440',
'SCORESBY',
'Mad',
```

(continues on next page)

(continued from previous page)

```
 'agonies',
 'endures',
 'infuriated',
 'rears',
 'snaps',
 'propelled',
 'observers',
 'opportunities',
 'habitudes',
 'BEALE',
 'offensively',
 'artful',
 'mischievous',
 'FREDERICK',
 'DEBELL',
 '1840',
 'October',
 'Raise',
 'ay',
 'THAR',
 'bowes',
 'os',
 'ROSS',
 'ETCHINGS',
 'CRUIZE',
 '1846',
 'Globe',
 'transactions',
 'relate',
 'HUSSEY',
 'SURVIVORS',
 'parried',
 'MISSIONARY',
 'JOURNAL',
 'TYERMAN',
 'boldest',
 'persevering',
 'REPORT',
 'DANIEL',
 'SPEECH',
 'SENATE',
 'APPLICATION',
 'ERECTION',
 'BREAKWATER',
 'CAPTORS',
 'WHALEMAN',
 'ADVENTURES',
 'BIOGRAPHY',
 'GATHERED',
 'HOMEWARD',
 'COMMODORE',
 'PREBLE',
 'REV',
 'CHEEVER',
 'MUTINEER',
 'BROTHER',
 'ANOTHER',
```

(continues on next page)

(continued from previous page)

```
 'MCCULLOCH',
 'COMMERCIAL',
 'reciprocal',
 'clews',
 'SOMETHING',
 'UNPUBLISHED',
 'CURRENTS',
 'Pedestrians',
 'recollect',
 'gateways',
 'VOYAGER',
 'ARCTIC',
 'NEWSPAPER',
 'TAKING',
 'RETAKING',
 'HOBOMACK',
 'MIRIAM',
 'FISHERMAN',
 'appliance',
 'RIBS',
 'TRUCKS',
 'Terra',
 'Del',
 'Fuego',
 'DARWIN',
 'NATURALIST',
 ";--'",
 '!\'"',
 'WHARTON',
 'Loomings',
 'spleen',
 'regulating',
 'circulation',
 'Whenever',
 'drizzly',
 'hypos',
 'philosophical',
 'Cato',
 'Manhattoes',
 'reefs',
 'downtown',
 'gazers',
 'Circumambulate',
 'Corlears',
 'Coenties',
 'Slip',
 'Whitehall',
 'Posted',
 'sentinels',
 'spiles',
 'pier',
 'lath',
 'counters',
 'desks',
 'loitering',
 'shady',
 'Inlanders',
```

(continues on next page)

(continued from previous page)

```
 'lanes',
 'alleys',
 'attract',
 'dale',
 'dreamiest',
 'shadiest',
 'quietest',
 'enchanting',
 'Saco',
 'crucifix',
 'Deep',
 'mazy',
 'Tiger',
 'Tennessee',
 'Rockaway',
 'Persians',
 'deity',
 'Narcissus',
 'ungraspable',
 'hazy',
 'quarrelsome',
 'offices',
 'abominate',
 'toils',
 'trials',
 'barques',
 'schooners',
 'broiling',
 'buttered',
 'judgmatically',
 'peppered',
 'reverentially',
 'idolatrous',
 'dotings',
 'ibis',
 'roasted',
 'bake',
 'plumb',
 'Van',
 'Rensselaers',
 'Randolphs',
 'Hardicanutes',
 'lording',
 'tallest',
 'decoction',
 'Seneca',
 'Stoics',
 'Testament',
 'promptly',
 'rub',
 'infliction',
 'BEING',
 'PAID',
 'urbane',
 'ills',
 'monied',
 'consign',
```

(continues on next page)

(continued from previous page)

```
 'prevalent',
 'violate',
 'Pythagorean',
 'commonalty',
 'police',
 'surveillance',
 'programme',
 'solo',
 'CONTESTED',
 'ELECTION',
 'PRESIDENCY',
 'UNITED',
 'STATES',
 'ISHMAEL',
 'BLOODY',
 'AFFGHANISTAN',
 'managers',
 'genteel',
 'comedies',
 'farces',
 'cunningly',
 'disguises',
 'cajoling',
 'unbiased',
 'freewill',
 'discriminating',
 'overwhelming',
 'undeliverable',
 'itch',
 'forbidden',
 'ignoring',
 'lodges',
 'Carpet',
 'Bag',
 'Manhatto',
 'candidates',
 'penalties',
 'Tyre',
 'Carthage',
 'imported',
 'cobblestones',
 'bitingly',
 'shouldering',
 'price',
 'fervent',
 'asphaltic',
 'pavement',
 'flinty',
 'projections',
 'soles',
 'Too',
 'cheapest',
 'cheeriest',
 'invitingly',
 'particles',
 'peer',
 'Angel',
```

(continues on next page)

(continued from previous page)

```
 'Doom',
 'wailing',
 'gnashing',
 'Wretched',
 'entertainment',
 'Moving',
 'emigrant',
 'poverty',
 'creak',
 'lodgings',
 'zephyr',
 'hob',
 'toasting',
 'observest',
 'sashless',
 'glazier',
 'reasonest',
 'chinks',
 'crannies',
 'lint',
 'chattering',
 'shiverings',
 'cob',
 'redder',
 'Orion',
 'glitters',
 'conservatories',
 'president',
 'temperance',
 'blubbering',
 'straggling',
 'wainscots',
 'reminding',
 'oilpainting',
 'besmoked',
 'defaced',
 'unequal',
 'crosslights',
 'hags',
 'delineate',
 'bewitched',
 'ponderings',
 'boggy',
 'soggy',
 'squitchy',
 'froze',
 'heath',
 'icebound',
 'represents',
 'Horner',
 'foundered',
 'clubs',
 'harvesting',
 'hacking',
 'horrifying',
 'Mixed',
 'Nathan',
```

(continues on next page)

(continued from previous page)

```
 'Swain',
 'corkscrew',
 'Blanco',
 'sojourning',
 'fireplaces',
 'duskier',
 'cockpits',
 'rarities',
 'Projecting',
 'Within',
 'shelves',
 'flasks',
 'bustles',
 'deliriums',
 'Abominable',
 'tumblers',
 'cylinders',
 'goggling',
 'deceitfully',
 'tapered',
 'Parallel',
 'pecked',
 'footpads',
 'Fill',
 'shilling',
 'examining',
 'SKRIMSHANDER',
 'accommodated',
 'unoccupied',
 'haint',
 'pose',
 'whalin',
 'decidedly',
 'objectionable',
 'wander',
 'Battery',
 'ruminating',
 'adorning',
 'potatoes',
 'sartainty',
 'diabolically',
 'steaks',
 'undress',
 'looker',
 'rioting',
 'Grampus',
 'seed',
 'Feegees',
 'tramping',
 'Enveloped',
 'bedarned',
 'eruption',
 'officiating',
 'brimmers',
 'complained',
 'potion',
 'colds',
```

(continues on next page)

(continued from previous page)

```
 'catarrhs',
 'liquor',
 'arrantest',
 'topers',
 'obstreperously',
 'aloof',
 'desirous',
 'hilarity',
 'coffer',
 'Southerner',
 'mountaineers',
 'Alleghanian',
 'missed',
 'supernaturally',
 'congratulate',
 'multiply',
 'bachelor',
 'abominated',
 'tidiest',
 'bedwards',
 'shan',
 'tablecloth',
 'Skrimshander',
 'bump',
 'spraining',
 'eider',
 'yoking',
 'rickety',
 'whirlwinds',
 'knockings',
 'dismissed',
 'popped',
 'cherishing',
 'chuckled',
 'chuckle',
 'mightily',
 'catches',
 'bamboozingly',
 'overstocked',
 'toothpick',
 'rayther',
 'BROWN',
 'slanderin',
 'farrago',
 'BROKE',
 'Sartain',
 'Mt',
 'Hecla',
 'persist',
 'mystifying',
 'unsay',
 'criminal',
 'Wall',
 'purty',
 'sarmon',
 'rips',
 'tellin',
```

(continues on next page)

(continued from previous page)

```
 'bought',
 'balmed',
 'curios',
 'sellin',
 'inions',
 'fooling',
 'idolators',
 'Depend',
 'reg',
 'lar',
 'spliced',
 'Johnny',
 'sprawling',
 'Arter',
 'glim',
 'jiffy',
 'irresolute',
 'vum',
 'WON',
 'Folding',
 'scrutiny',
 'porcupine',
 'moccasin',
 'ponchos',
 'parade',
 'rainy',
 'remembering',
 'commended',
 'cobs',
 'Nod',
 'footfall',
 'unlacing',
 'blackish',
 'plasters',
 'inkling',
 'Placing',
 'crammed',
 'scalp',
 'mildewed',
 'Ignorance',
 'parent',
 'nonplussed',
 'undressing',
 'checkered',
 'Thirty',
 'frogs',
 'quaked',
 'wrapall',
 'dreadnaught',
 'fumbled',
 'Remembering',
 'manikin',
 'tenpin',
 'andirons',
 'jambs',
 'bricks',
 'appropriate',
```

(continues on next page)

(continued from previous page)

```
 'applying',
 'hastier',
 'withdrawals',
 'antics',
 'devotee',
 'extinguishing',
 'unceremoniously',
 'bagged',
 'sportsman',
 'woodcock',
 'uncomfortableness',
 'deliberating',
 'puffed',
 'sang',
 'Stammering',
 'conjured',
 'responses',
 'debel',
 'flourishing',
 'Angels',
 'flourishings',
 'peddlin',
 'sleepe',
 'grunted',
 'gettee',
 'motioning',
 'comely',
 'insured',
 'Counterpane',
 'parti',
 'triangles',
 'interminable',
 'caper',
 'supperless',
 '21st',
 'hemisphere',
 'sigh',
 'Sixteen',
 'ached',
 'coaches',
 'stockinged',
 'slippering',
 'misbehaviour',
 'unendurable',
 'stepmothers',
 'misfortunes',
 'steeped',
 'shudderingly',
 'confounding',
 'soberly',
 'recurred',
 'predicament',
 'unlock',
 'bridegroom',
 'clasp',
 'hugged',
 'rouse',
```

(continues on next page)

(continued from previous page)

```
 'snore',
 'scratch',
 'Throwing',
 'expostulations',
 'unbecomingness',
 'matrimonial',
 'dawning',
 'overture',
 'innate',
 'compliment',
 'civility',
 'rudeness',
 'toilette',
 'dressing',
 'donning',
 'gaspings',
 'booting',
 'caterpillar',
 'outlandishness',
 'manners',
 'education',
 'undergraduate',
 'dreamt',
 'cowhide',
 'pinched',
 'curtains',
 'indecorous',
 'contented',
 'restricting',
 'donned',
 'lathering',
 'unsheathes',
 'whets',
 'Rogers',
 'cutlery',
 'Afterwards',
 'baton',
 'Breakfast',
 'pleasantly',
 'bountifully',
 'laughable',
 'bosky',
 'unshorn',
 'gowns',
 'toasted',
 'lingers',
 'tarried',
 'barred',
 'Grub',
 'Park',
 'assurance',
 'polish',
 'occasioned',
 'embarrassed',
 'bashfulness',
 'duelled',
 'winking',
```

(continues on next page)

(continued from previous page)

```
 'tastes',
 'sheepishly',
 'bashful',
 'icicle',
 'admirer',
 'cordially',
 'grappling',
 'genteelly',
 'eschewed',
 'undivided',
 '6',
 'circulating',
 'nondescripts',
 'Chestnut',
 'jostle',
 'Regent',
 'Lascars',
 'Bombay',
 'Apollo',
 'Feegeeans',
 'Tongatobooarrs',
 'Erromanggoans',
 'Pannangians',
 'Brighggians',
 'weekly',
 'Vermonters',
 'stalwart',
 'frames',
 'felled',
 'strutting',
 'wester',
 'bombazine',
 'cloak',
 'mow',
 'gloves',
 'joins',
 'outfit',
 'waistcoats',
 'Hay',
 'Seed',
 'tract',
 'dearest',
 'pave',
 'eggs',
 'patrician',
 'parks',
 'scraggy',
 'scoria',
 'Herr',
 'dowers',
 'nieces',
 'reservoirs',
 'maples',
 'bountiful',
 'proffer',
 'passer',
 'cones',
```

(continues on next page)

(continued from previous page)

```
 'blossoms',
 'superinduced',
 'carnation',
 'Salem',
 'sweethearts',
 'Puritanic',
 'Whaleman',
 'Wrapping',
 'Each',
 'quote',
 'TALBOT',
 'Near',
 'Desolation',
 '1st',
 'SISTER',
 'ROBERT',
 'WILLIS',
 'ELLERY',
 'NATHAN',
 'COLEMAN',
 'WALTER',
 'CANNY',
 'SETH',
 'GLEIG',
 'Forming',
 'ELIZA',
 '31st',
 'MARBLE',
 'SHIPMATES',
 'EZEKIEL',
 'HARDY',
 'AUGUST',
 '3d',
 '1833',
 'WIDOW',
 'Shaking',
 'glazed',
 'Affected',
 'relatives',
 'unhealing',
 'sympathetically',
 'wounds',
 'bleed',
 'blanks',
 ...]
```

2.3.2 Fine-grained Selection of Words

Next, let's look at the long words of a text; perhaps these will be more characteristic and informative. For this we adapt some notation from set theory. We would like to find the words from the vocabulary of the text that are more than 15 characters long. Let's call this property `P`, so that `P(w)` is true if and only if `w` is more than 15 characters long. Now we can express the words of interest using mathematical set notation as shown in (2.1). This means "the set of all `w` such that `w` is an element of `V` (the vocabulary) and `w` has property `P`".

$$w|w \in VP(w) \tag{2.1}$$

[w for w in V if p(w)]

The corresponding Python expression is given in

```
[w for w in V if p(w)]
```

Note that it produces a list, not a set, which means that duplicates are possible. Observe how similar the two notations are. Let's go one more step and write executable Python code:

```
V = set(text1)
long_words = [w for w in V if len(w) > 15]
sorted(long_words)
```

```
['CIRCUMNAVIGATION',
 'Physiognomically',
 'apprehensiveness',
 'cannibalistically',
 'characteristically',
 'circumnavigating',
 'circumnavigation',
 'circumnavigations',
 'comprehensiveness',
 'hermaphroditical',
 'indiscriminately',
 'indispensableness',
 'irresistibleness',
 'physiognomically',
 'preternaturalness',
 'responsibilities',
 'simultaneousness',
 'subterraneousness',
 'supernaturalness',
 'superstitiousness',
 'uncomfortableness',
 'uncompromisedness',
 'undiscriminating',
 'uninterpenetratingly']
```

Your Turn

Try out the previous statements in the Jupyter Notebook, and experiment with changing the text and changing the length condition. Does it make a difference to your results if you change the variable names, e.g., using `[word for word in vocab if ...]`?

Let's return to our task of finding words that characterize a text. Notice that the long words in `text4` reflect its national focus — constitutionally, transcontinental — whereas those in `text5` reflect its informal content: boooooooooooglyyyyyy

and yuuuuuuuuuuuummmmmmmmmmmm. Have we succeeded in automatically extracting words that typify a text? Well, these very long words are often *hapaxes* (i.e., unique) and perhaps it would be better to find frequently occurring long words. This seems promising since it eliminates frequent short words (e.g., the) and infrequent long words (e.g. antiphilosophists). Here are all words from the chat corpus that are longer than seven characters, that occur more than seven times:

```
fdist5 = FreqDist(text5)
sorted(w for w in set(text5) if len(w) > 7 and fdist5[w] > 7)
```

```
['#14-19teens',
 '#talkcity_adults',
 '((((((((((',
 '........',
 'Question',
 'actually',
 'anything',
 'computer',
 'cute.-ass',
 'everyone',
 'football',
 'innocent',
 'listening',
 'remember',
 'seriously',
 'something',
 'together',
 'tomorrow',
 'watching']
```

Notice how we have used two conditions: `len(w) > 7` ensures that the words are longer than seven letters, and `fdist5[w] > 7` ensures that these words occur more than seven times. At last we have managed to automatically identify the frequently-occurring content-bearing words of the text. It is a modest but important milestone: a tiny piece of code, processing tens of thousands of words, produces some informative output.

2.3.3 Collocations and Bigrams

A collocation is a sequence of words that occur together unusually often. Thus `red wine` is a collocation, whereas `the wine` is not. A characteristic of collocations is that they are resistant to substitution with words that have similar senses; for example, `maroon wine` sounds definitely odd.

To get a handle on collocations, we start off by extracting from a text a list of word pairs, also known as bigrams. This is easily accomplished with the function `bigrams()`:

```
list(bigrams(['more', 'is', 'said', 'than', 'done']))
```

```
[('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]
```

If you omitted list() above, and just typed bigrams([‘more’, …]), you would have seen output of the form <generator object bigrams at 0x10fb8b3a8>. This is Python’s way of saying that it is ready to compute a sequence of items, in this case, bigrams. For now, you just need to know to tell Python to convert it into a list, using list().

Here we see that the pair of words `than-done` is a bigram, and we write it in Python as (‘than’, ‘done’). Now, collocations are essentially just frequent bigrams, except that we want to pay more attention to the cases that involve rare words. In particular, we want to find bigrams that occur more often than we would expect based on the frequency of the individual words. The collocations() function does this for us. We will see how it works later.

```
text4.collocations()
```

```
United States; fellow citizens; four years; years ago; Federal
Government; General Government; American people; Vice President; God
bless; Chief Justice; Old World; Almighty God; Fellow citizens; Chief
Magistrate; every citizen; one another; fellow Americans; Indian
tribes; public debt; foreign nations
```

```
text8.collocations()
```

```
would like; medium build; social drinker; quiet nights; non smoker;
long term; age open; Would like; easy going; financially secure; fun
times; similar interests; Age open; weekends away; poss rship; well
presented; never married; single mum; permanent relationship; slim
build
```

The collocations that emerge are very specific to the genre of the texts. In order to find red wine as a collocation, we would need to process a much larger body of text.

2.3.4 Counting Other Things

Counting words is useful, but we can count other things too. For example, we can look at the distribution of word lengths in a text, by creating a FreqDist out of a long list of numbers, where each number is the length of the corresponding word in the text:

```
fdist = FreqDist(len(w) for w in text1)
print(fdist)
```

```
<FreqDist with 19 samples and 260819 outcomes>
```

```
fdist
```

```
FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111, 7: 14399, 8:␣
↪9966, 9: 6428, 10: 3528, ...})
```

We start by deriving a list of the lengths of words in text1, and the FreqDist then counts the number of times each of these occurs. The result is a distribution containing a quarter of a million items, each of which is a number corresponding to a word token in the text. But there are at most only 20 distinct items being counted, the numbers 1 through 20, because there are only 20 different word lengths. That is, there are words consisting of just one character, two characters, ..., twenty characters, but none with twenty one or more characters. One might wonder how frequent the different lengths of word are (e.g., how many words of length four appear in the text, are there more words of length five than length four, etc). We can do this as follows:

```
fdist.most_common()
```

```
[(3, 50223),
 (1, 47933),
 (4, 42345),
 (2, 38513),
 (5, 26597),
 (6, 17111),
 (7, 14399),
```

(continues on next page)

(continued from previous page)

```
 (8, 9966),
 (9, 6428),
 (10, 3528),
 (11, 1873),
 (12, 1053),
 (13, 567),
 (14, 177),
 (15, 70),
 (16, 22),
 (17, 12),
 (18, 1),
 (20, 1)]
```

```
fdist.max()
```

```
3
```

```
fdist[3]
```

```
50223
```

```
fdist.freq(3)
```

```
0.19255882431878046
```

From this we see that the most frequent word length is 3, and that words of length 3 account for roughly 50,000 (or 20%) of the words making up the book. Although we will not pursue it here, further analysis of word length might help us understand differences between authors, genres, or languages.

3.1 summarizes the functions defined in frequency distributions.

Table 2.2: Functions Defined for NLTK's Frequency Distributions

Example	Description
fdist = FreqDist(samples)	create a frequency distribution containing the given samples
fdist[sample] += 1	increment the count for this sample
fdist['monstrous']	count of the number of times a given sample occurred
fdist.freq('monstrous')	frequency of a given sample
fdist.N()	total number of samples
fdist.most_common(n)	the n most common samples and their frequencies
for sample in fdist:	iterate over the samples
fdist.max()	sample with the greatest count
fdist.tabulate()	tabulate the frequency distribution
fdist.plot()	graphical plot of the frequency distribution
fdist.plot(cumulative=True)	cumulative plot of the frequency distribution
fdist1 |= fdist2	update fdist1 with counts from fdist2
fdist1 < fdist2	test if samples in fdist1 occur less frequently than in fdist2

Our discussion of frequency distributions has introduced some important Python concepts, and we will look at them systematically.

2.4 Back to Python: Making Decisions and Taking Control

So far, our little programs have had some interesting qualities: the ability to work with language, and the potential to save human effort through automation. A key feature of programming is the ability of machines to make decisions on our behalf, executing instructions when certain conditions are met, or repeatedly looping through text data until some condition is satisfied. This feature is known as control, and is the focus of this section.

4.1 Conditionals Python supports a wide range of operators, such as < and >=, for testing the relationship between values. The full set of these relational operators is shown in 4.1.

Table 2.3: Numerical Comparison Operators

Operator	Relationship
<	less than
<=	less than or equal to
==	equal to (note this is two "=" signs, not one)
!=	not equal to
>	greater than
>=	greater than or equal to

We can use these to select different words from a sentence of news text. Here are some examples — only the operator is changed from one line to the next. They all use `sent7`, the first sentence from text7 (Wall Street Journal). As before, if you get an error saying that `sent7` is undefined, you need to first type: `from nltk.book import *`

```
from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

```
sent7
```

```
['Pierre',
 'Vinken',
 ',',
 '61',
 'years',
 'old',
 ',',
 'will',
 'join',
 'the',
 'board',
 'as',
 'a',
```

(continues on next page)

(continued from previous page)

```
 'nonexecutive',
 'director',
 'Nov.',
 '29',
 '.']
```

```
[w for w in sent7 if len(w) < 4]
```

```
[',', '61', 'old', ',', 'the', 'as', 'a', '29', '.']
```

```
[w for w in sent7 if len(w) <= 4]
```

```
[',', '61', 'old', ',', 'will', 'join', 'the', 'as', 'a', 'Nov.', '29', '.']
```

```
[w for w in sent7 if len(w) == 4]
```

```
['will', 'join', 'Nov.']
```

```
[w for w in sent7 if len(w) != 4]
```

```
['Pierre',
 'Vinken',
 ',',
 '61',
 'years',
 'old',
 ',',
 'the',
 'board',
 'as',
 'a',
 'nonexecutive',
 'director',
 '29',
 '.']
```

There is a common pattern to all of these examples: `[w for w in text if condition ]`, where condition is a Python "test" that yields either true or false. In the cases shown in the previous code example, the condition is always a numerical comparison.

Instead of writing your own Regex, we can test various properties of words, using the functions for a string object listed below.

Table 2.4: Some Word Comparison Operators

Function	Meaning
s.startswith(t)	test if s starts with t
s.endswith(t)	test if s ends with t
t in s	test if t is a substring of s
s.islower()	test if s contains cased characters and all are lowercase
s.isupper()	test if s contains cased characters and all are uppercase
s.isalpha()	test if s is non-empty and all characters in s are alphabetic
s.isalnum()	test if s is non-empty and all characters in s are alphanumeric
s.isdigit()	test if s is non-empty and all characters in s are digits
s.istitle()	test if s contains cased characters and is titlecased (i.e. all words in s have initial capitals)

Here are some examples of these operators being used to select words from our texts: words ending with `-ableness`; words containing `gnt`; words having an initial capital; and words consisting entirely of digits.

```
sorted(w for w in set(text1) if w.endswith('ableness'))
```

```
['comfortableness',
 'honourableness',
 'immutableness',
 'indispensableness',
 'indomitableness',
 'intolerableness',
 'palpableness',
 'reasonableness',
 'uncomfortableness']
```

```
sorted(term for term in set(text4) if 'gnt' in term)
```

```
['Sovereignty', 'sovereignties', 'sovereignty']
```

```
sorted(item for item in set(text6) if item.istitle())[-5:]
```

```
['Yes', 'You', 'Your', 'Yup', 'Zoot']
```

```
sorted(item for item in set(sent7) if item.isdigit())
```

```
['29', '61']
```

We can also create more complex conditions. If `c` is a condition, then `not c` is also a condition. If we have two conditions `c1` and `c2`, then we can combine them to form a new condition using conjunction and disjunction: `c1 and c2`, `c1 or c2`.

Your Turn

Run the following examples and try to explain what is going on in each one. Next, try to make up some conditions of your own.

```
sorted(w for w in set(text7) if '-' in w and 'index' in w)
sorted(wd for wd in set(text3) if wd.istitle() and len(wd) > 10)
```

(continues on next page)

(continued from previous page)

```
sorted(w for w in set(sent7) if not w.islower())
sorted(t for t in set(text2) if 'cie' in t or 'cei' in t)
```

2.4.1 Operating on Every Element

We saw some examples of counting items other than words. Let's take a closer look at the notation we used:

Note: These expressions have the form [f(w) for …] or [w.f() for …], where f is a function that operates on a word to compute its length, or to convert it to uppercase. For now, you don't need to understand the difference between the notations f(w) and w.f(). Instead, simply learn this Python idiom which performs the same operation on every element of a list. In the preceding examples, it goes through each word in text1, assigning each one in turn to the variable w and performing the specified operation on the variable.

The notation just described is called a **list comprehension**." This is a Python idiom, a fixed notation that we use habitually without bothering to analyze each time. Mastering such idioms is an important part of becoming a fluent Python programmer.

Let's return to the question of vocabulary size, and apply the same idiom here:

```
len(text1)
```

```
260819
```

```
len(set(text1))
```

```
len(set(word.lower() for word in text1))
```

```
17231
```

Now that we are not double-counting words like This and this, which differ only in capitalization, we've wiped 2,000 off the vocabulary count! We can go a step further and eliminate numbers and punctuation from the vocabulary count by filtering out any non-alphabetic items:

```
len(set(word.lower() for word in text1 if word.isalpha()))
```

```
16948
```

This example is slightly complicated: it lowercases all the purely alphabetic items. Perhaps it would have been simpler just to count the lowercase-only items, but this gives the wrong answer (why?).

2.4.2 Nested Code Blocks

Most programming languages permit us to execute a block of code when a conditional expression, or if statement, is satisfied. We already saw examples of conditional tests in code like `[w for w in sent7 if len(w) < 4]`. In the following program, we have created a variable called word containing the string value 'cat'. The if statement checks whether the test `len(word) < 5` is true. It is, so the body of the if statement is invoked and the print statement is executed, displaying a message to the user. Remember to indent the print statement by typing four spaces.

```
word = 'cat'
if len(word) < 5:
    print('word length is less than 5')
else:
    print('word length is greater than or equal to 5')
```

```
word length is less than 5
```

An if statement is known as a control structure because it controls whether the code in the indented block will be run. Another control structure is the `for` loop. Try the following, and remember to include the colon and the four spaces:

```
for word in ['Call', 'me', 'Ishmael', '.']:
   print(word)
```

```
Call
me
Ishmael
.
```

This is called a loop because Python executes the code in circular fashion. It starts by performing the assignment word = 'Call', effectively using the word variable to name the first item of the list. Then, it displays the value of word to the user. Next, it goes back to the for statement, and performs the assignment word = 'me', before displaying this new value to the user, and so on. It continues in this fashion until every item of the list has been processed.

Tip: If we want to create a list by iterating through a list, "list comprehension" is a Pythonic way and preferred than a loop.

2.4.3 Looping with Conditions

Now we can combine the if and for statements. We will loop over every item of the list, and print the item only if it ends with the letter l. We'll pick another name for the variable to demonstrate that Python doesn't try to make sense of variable names.

```
sent1 = ['Call', 'me', 'Ishmael', '.']
for xyzzy in sent1:
    if xyzzy.endswith('l'):
        print(xyzzy)
```

```
Call
Ishmael
```

You will notice that if and for statements have a colon at the end of the line, before the indentation begins. In fact, all Python control structures end with a colon. The colon indicates that the current statement relates to the indented block that follows.

We can also specify an action to be taken if the condition of the if statement is not met. Here we see the elif (else if) statement, and the else statement. Notice that these also have colons before the indented code.

```
for token in sent1:
    if token.islower():
        print(token, 'is a lowercase word')
    elif token.istitle():
```

(continues on next page)

(continued from previous page)

```
        print(token, 'is a titlecase word')
    else:
        print(token, 'is punctuation')
```

```
Call is a titlecase word
me is a lowercase word
Ishmael is a titlecase word
. is punctuation
```

As you can see, even with this small amount of Python knowledge, you can start to build multiline Python programs. It's important to develop such programs in pieces, testing that each piece does what you expect before combining them into a program. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

Finally, let's combine the idioms we've been exploring. First, we create a list of cie and cei words, then we loop over each item and print it. Notice the extra information given in the print statement: end=' '. This tells Python to print a space (not the default newline) after each word.

```
tricky = sorted(w for w in set(text2) if 'cie' in w or 'cei' in w)
for word in tricky:
    print(word, end=' ')
```

```
ancient ceiling conceit conceited conceive conscience conscientious conscientiously␣
↪deceitful deceive deceived deceiving deficiencies deficiency deficient delicacies␣
↪excellencies fancied insufficiency insufficient legacies perceive perceived␣
↪perceiving prescience prophecies receipt receive received receiving society species␣
↪sufficient sufficiently undeceive undeceiving
```

2.5 Exercises

1. Try using the Python interpreter as a calculator, and typing expressions like `12 / (4 + 1)`.
2. Given an alphabet of 26 letters, there are 26 to the power 10, or `26 ** 10`, ten-letter strings we can form. That works out to 141167095653376. How many hundred-letter strings are possible?
3. The Python multiplication operation can be applied to lists. What happens when you type `['Monty', 'Python'] * 20`, or `3 * sent1`?
4. How many words are there in `text2`? How many distinct words are there?
5. Compare the lexical diversity scores for humor and romance fiction in 1.1. Which genre is more lexically diverse?
6. Produce a dispersion plot of the four main protagonists in `Sense and Sensibility`: `Elinor`, `Marianne`, `Edward`, and `Willoughby`. What can you observe about the different roles played by the males and females in this novel? Can you identify the couples?
7. Find the collocations in text5.
8. Consider the following Python expression: `len(set(text4))`. State the purpose of this expression. Describe the two steps involved in performing this computation.
9. Define a string and assign it to a variable, e.g., `my_string = 'My String'` (but put something more interesting in the string).
    - Print the contents of this variable in two ways, first by simply typing the variable name and pressing enter, then by using the print statement.

- Try adding the string to itself using `my_string + my_string`, or multiplying it by a number, e.g.,`my_string * 3`.
- Notice that the strings are joined together without any spaces. How could you fix this?

10. Define a variable `my_sent` to be a list of words, using the syntax `my_sent = ["My", "sent"]` (but with your own words, or a favorite saying).
    - Use `' '.join(my_sent)` to convert this into a string.
    - Use split() to split the string back into the list form you had to start with.
11. Define several variables containing lists of words, e.g., `phrase1`, `phrase2`, and so on. Join them together in various combinations (using the plus operator) to form whole sentences. What is the relationship between `len(phrase1 + phrase2)` and `len(phrase1) + len(phrase2)`?
12. Consider the following two expressions, which have the same value. Which one will typically be more relevant in NLP? Why?

```
"Monty Python"[6:12]
["Monty", "Python"][1]
```

13. We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `sent1[2][2]` do? Why? Experiment with other index values.
14. The first sentence of `text3` is provided to you in the variable sent3. The index of `the` in `sent3` is 1, because `sent3[1]` gives us 'the'. What are the indexes of the two other occurrences of this word in `sent3`?
15. Find all words in the `Chat Corpus` (`text5`) starting with the letter `b`. Show them in alphabetical order.
16. Type the expression `list(range(10))` at the interpreter prompt. Now try `list(range(10, 20))`, `list(range(10, 20, 2))`, and `list(range(20, 10, -2))`. We will see a variety of uses for this built-in function in later part of the unit.
17. Use `text9.index()` to find the index of the word `sunset`. You'll need to insert this word as an argument between the parentheses. By a process of trial and error, find the slice for the complete sentence that contains this word.
    - Using list addition, and the set and sorted operations, compute the vocabulary of the sentences `sent1 ... sent8`.
18. What is the difference between the following two lines? Which one will give a larger value? Will this be the case for other texts?

```
sorted(set(w.lower() for w in text1))
sorted(w.lower() for w in set(text1))
```

19. What is the difference between the following two tests: `w.isupper()` and `not w.islower()`?
20. Write the slice expression that extracts the last two words of `text2`.
21. Find all the four-letter words in `the Chat Corpus` (`text5`). With the help of a frequency distribution (FreqDist), show these words in decreasing order of frequency.
22. Review the discussion of looping with conditions. Use a combination of `for` and `if` statements to loop over the words of the movie script for `Monty Python and the Holy Grail` (`text6`) and print all the uppercase words, one per line.
23. Write expressions for finding all words in `text6` that meet the conditions listed below. The result should be in the form of a list of words: `['word1', 'word2', ...]`.
    - Ending in ise
    - Containing the letter z

- Containing the sequence of letters pt
- Having all lowercase letters except for an initial capital (i.e., titlecase)

24. Define sent to be the list of words ['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']. Now write code to perform the following tasks:
    - Print all words beginning with sh
    - Print all words longer than four characters
25. What does the following Python code do? `sum(len(w) for w in text1)` Can you use it to work out the average word length of a text?
26. Define a function called `vocab_size(text)` that has a single parameter for the text, and which returns the vocabulary size of the text.
27. Define a function percent(word, text) that calculates how often a given word occurs in a text, and expresses the result as a percentage.
28. We have been using sets to store vocabularies. Try the following Python expression: `set(sent3) < set(text1)`. Experiment with this using different arguments to `set()`. What does it do? Can you think of a practical application for this?

CHAPTER

THREE

LAB03: SPACY NLP PIPELINES

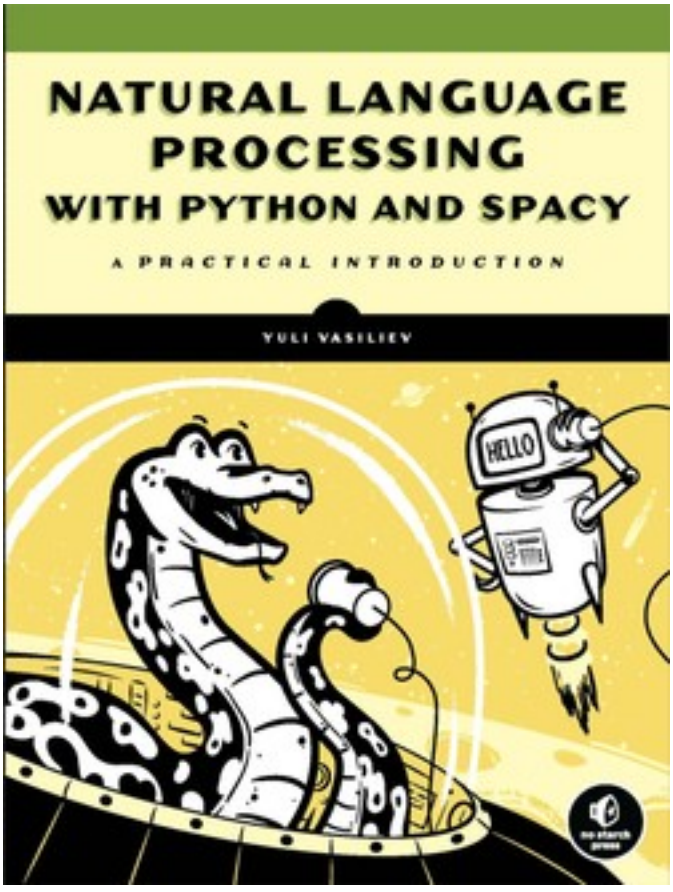
spaCy makes use of two core types of objects to support various NLP tasks.

A **container** object in spaCy groups multiple elements into a single unit. It can be a collection of objects, like tokens or sentences, or a set of annotations related to a single object.

Pipeline components objects that process the text input to create containers and fill them with relevant data, such as a part-of-speech tagger, a dependency parser and an entity recogniser.

In this lab, we will look at these two types of objects in spaCy to get a more in-depth understanding of how spaCy NLP code works. The we will test out our information extraction skills by deploying a simple rule-based chatbot.

Reference: (The book code works with spaCy v2.2, our lab code is compiled on spaCy v3.0.5)



3.1 Container Objects in spaCy

Container objects in spaCy mimic the structure of natural language texts: a text is composed of sentences, and each sentence contains tokens. Token, Span, and Doc, the most widely used container objects in spaCy from a user's standpoint, represent a token, a phrase or sentence, and a text, respectively. A container can contain other containers - for example, a Doc contains Tokens. In this section, we'll explore working with these container objects.

3.1.1 Doc

The `Doc()` constructor, requires two parameters:

- a vocab object, which is a storage container that provides vocabulary data, such as lexical types (adjective, verb, noun …);
- a list of tokens to add to the Doc object being created.

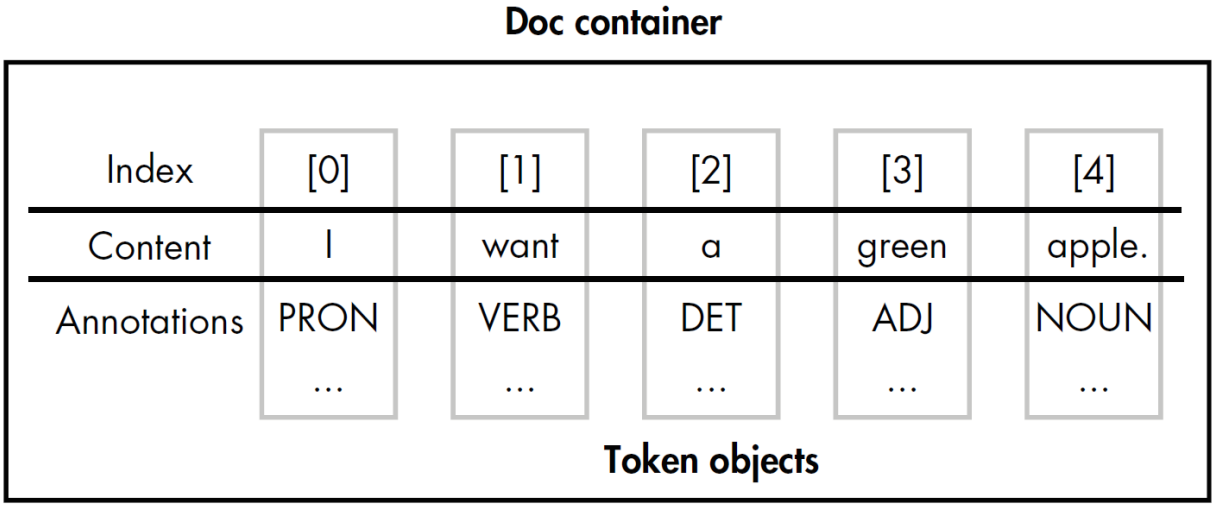
```
from spacy.tokens.doc import Doc
from spacy.vocab import Vocab

"""
create a spacy.tokens.doc.Doc object
using its constructor
"""
doc = Doc(Vocab(), words = [u'Hello', u'World!'])
print(doc)
print(type(doc))
```

```
Hello World!
<class 'spacy.tokens.doc.Doc'>
```

3.1.2 Token

spaCy's Token object is a container for a set of annotations related to a single token, such as that token's part of speech.

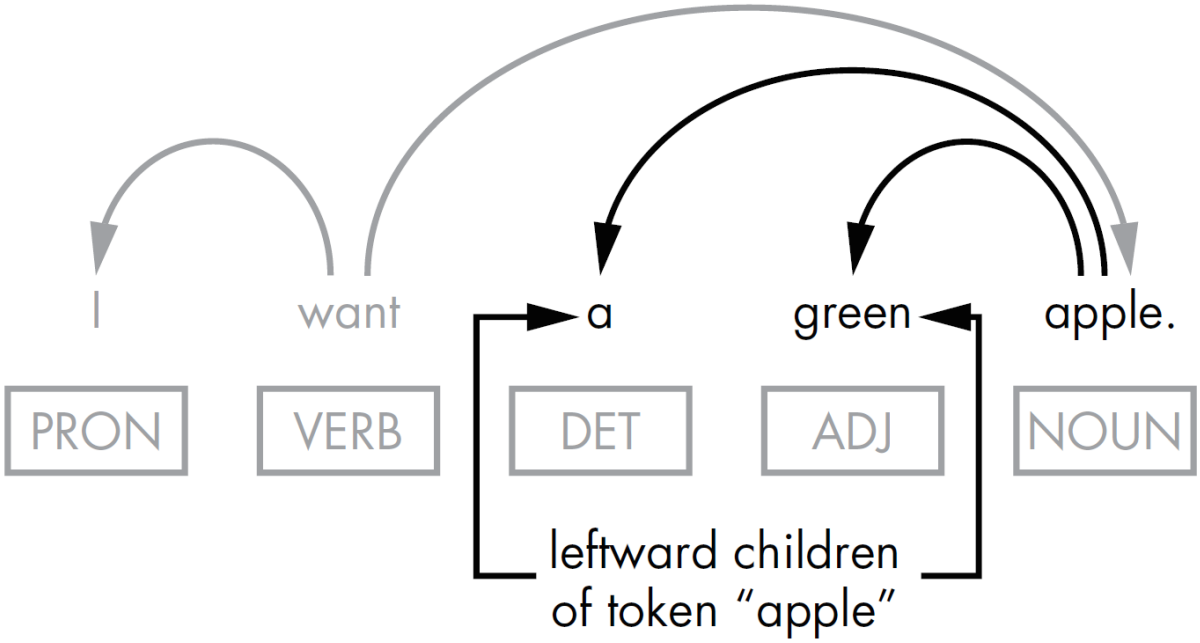
A Doc object contains a collection of the Token objects generated as a result of the tokenization performed on a submitted text. These tokens have indices, allowing you to access them based on their positions in the text.




```
import spacy
nlp = spacy.load('en_core_web_sm')
doc = nlp(u'I want a green apple.')
# token_text1 and token_text2 produce the same results
token_text1 = [token.text for token in doc]
token_text2 = [doc[i].text for i in range(len(doc))]
print(token_text1)
print(token_text2)
```

```
['I', 'want', 'a', 'green', 'apple', '.']
['I', 'want', 'a', 'green', 'apple', '.']
```

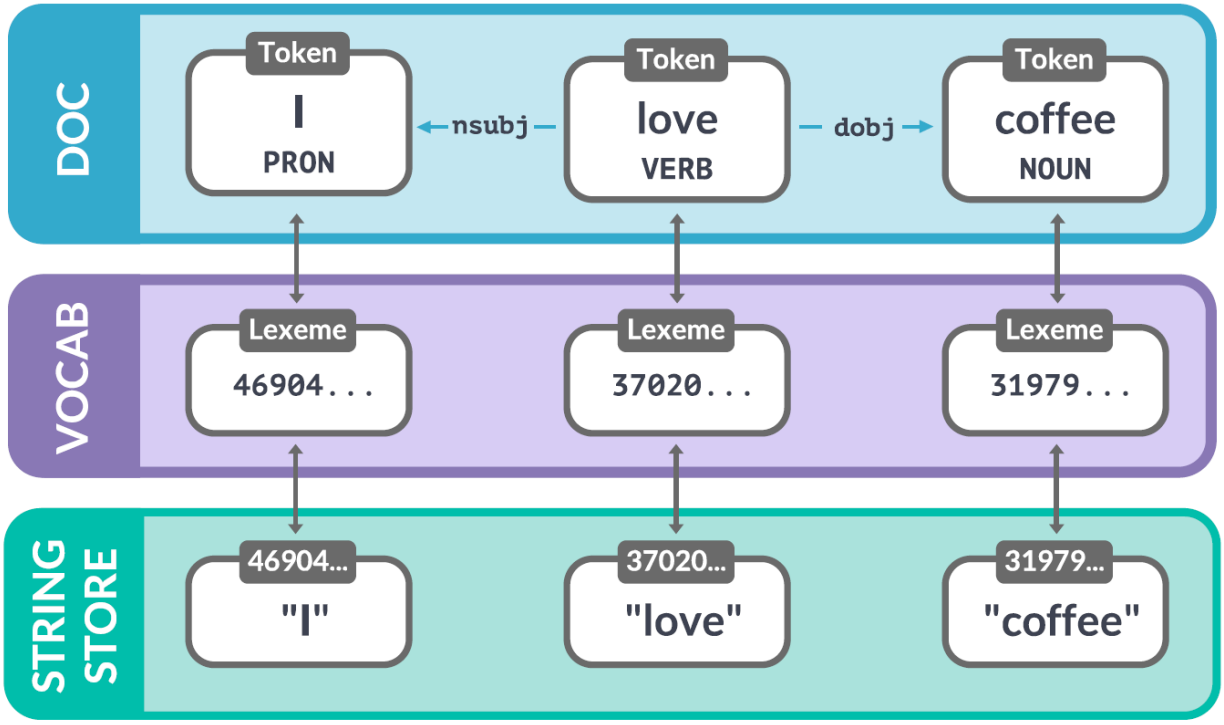
3.1.3 `Token.lefts` `Token.rights` and `Token.children`




```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp(u'I want a green apple.')
print([t for t in doc[4].lefts])
print([t for t in doc[4].children])
print([t for t in doc[1].rights])
```

```
[a, green]
[a, green]
[apple, .]
```

3.1.4 Vocab



- Whenever possible, spaCy tries to store data in a vocabulary, the Vocab storage class, that will be shared by multiple documents;
- To save memory, spaCy also encodes all strings to hash values. For example, "coffee" has the hash 3197928453018144401.
- Entity labels like "ORG" and part-of-speech tags like "VERB" are also encoded.

spaCy 101

```
import spacy
nlp = spacy.load('en_core_web_sm')
doc = nlp('I love coffee!')
for token in doc:
    lexeme = doc.vocab[token.text]
    print(lexeme.text, lexeme.orth, lexeme.shape_,
          lexeme.prefix_, lexeme.suffix_, lexeme.is_alpha,
          lexeme.is_digit, lexeme.is_title, lexeme.lang_)

print(doc.vocab.strings["coffee"]) # 3197928453018144401
print(doc.vocab.strings[3197928453018144401]) # 'coffee'
```

```
I 4690420944186131903 X I I True False True en
love 3702023516439754181 xxxx l ove True False False en
coffee 3197928453018144401 xxxx c fee True False False en
! 17494803046312582752 ! ! ! False False False en
3197928453018144401
coffee
```

3.1.5 Span

Span can be obtained as simple as doc[start:end] where start and end are the index of starting token and the ending token, respectively. The two indices can be

- manually specified; or
- computed through pattern matching

```
import spacy
from spacy.matcher import Matcher
from spacy.tokens import Doc, Span, Token
nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)
# A dependency label pattern that matches a word sequence
pattern = [{"DEP": "nsubj"},{"DEP": "aux"},{"DEP": "ROOT"}]
matcher.add("NsubjAuxRoot", [pattern])
doc = nlp(u"We can overtake them.")
# 1. Return (match_id, start, end) tuples
matches = matcher(doc)
for match_id, start, end in matches:
    span = doc[start:end]
    print("Span: ", span.text)
    print("The positions in the doc are: ", start, "-", end)
# 2. Return Span objects directly
matches = matcher(doc, as_spans=True)
for span in matches:
    print(span.text, span.label_)
```

```
Span:  We can overtake
The positions in the doc are:  0 - 3
We can overtake NsubjAuxRoot
```

3.1.6 Doc.noun_chunks and Retokenising

A **noun chunk** is a phrase that has a noun as its head.

```
import spacy
nlp = spacy.load('en_core_web_sm')

doc = nlp(u'The Golden Gate Bridge is an iconic landmark in San Francisco.')

# Retokenize to treat each noun_chunk as a single token
with doc.retokenize() as retokenizer:
  for chunk in doc.noun_chunks:
    print(chunk.text + ' ' + str(type(chunk)))
    retokenizer.merge(chunk)
    #doc.retokenize().merge(chunk)

for token in doc:
  print(token)
```

```
The Golden Gate Bridge <class 'spacy.tokens.span.Span'>
an iconic landmark <class 'spacy.tokens.span.Span'>
San Francisco <class 'spacy.tokens.span.Span'>
The Golden Gate Bridge
```

(continues on next page)

(continued from previous page)

```
is
an iconic landmark
in
San Francisco
.
```

3.1.7 Doc.sents

the Doc object's doc.sents property is an iterator over the sentences in a Doc object. For this reason, you can't use this property to refer to sentences by index, but you can iterate over them in a loop or create a list of Span objects where each span represents a sentence.

- Doc object's `doc.sents` property is an `generator` object, i.e. an iterator over the sentences in a Doc object. You can use for each in loop, but not subset indexing.
- Each member of the generator object is a Span of type spacy.tokens.span.Span.

Tip

`spans = list(doc.sents)` will return a list of span objects that each represent a sentence.

```
import spacy
nlp = spacy.load('en_core_web_sm')
doc = nlp(u'A storm hit the beach. It started to rain.')
for sent in doc.sents:
  print(type(sent))
  # Sentence level index
  [sent[i] for i in range(len(sent))]
# Doc level index
[doc[i] for i in range(len(doc))]
```

```
<class 'spacy.tokens.span.Span'>
<class 'spacy.tokens.span.Span'>
```

```
[A, storm, hit, the, beach, ., It, started, to, rain, .]
```

3.2 NLP Pipelines

3.2.1 Traditional NLP Pipeline in NLTK

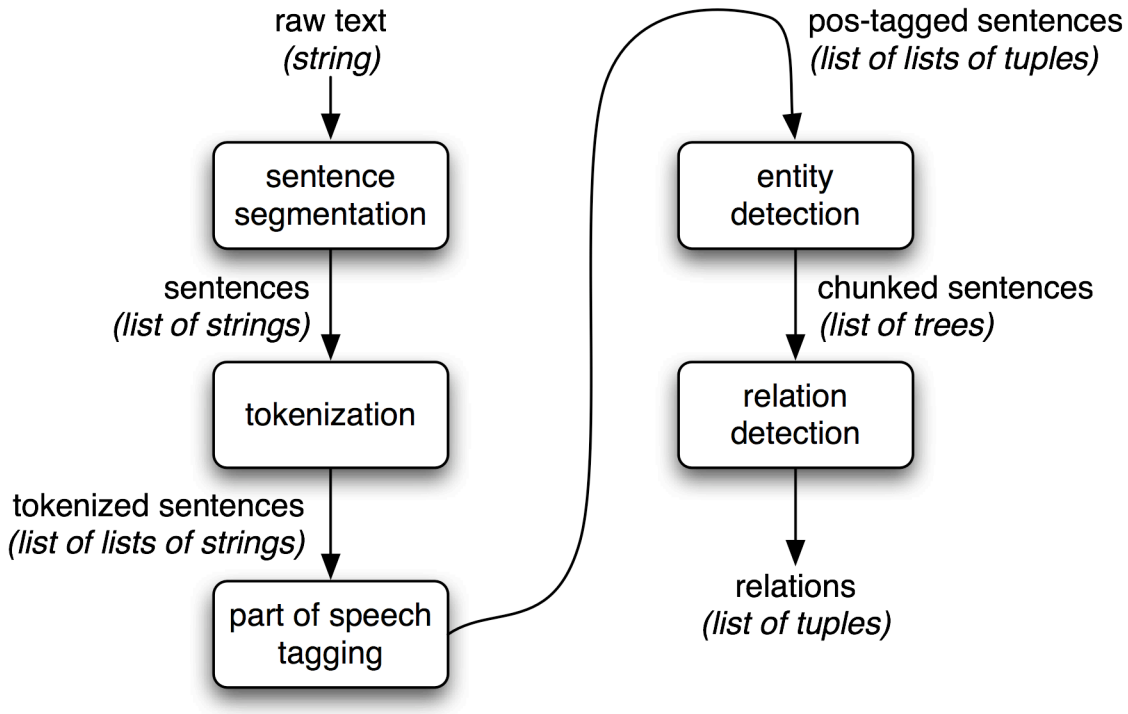


Image credit of Information Extraction in NLTK

```
import nltk
nltk.download('punkt') # Sentence Tokenize
nltk.download('averaged_perceptron_tagger') # POS Tagging
nltk.download('maxent_ne_chunker') # Named Entity Chunking
nltk.download('words') # Word Tokenize

# texts is a collection of documents.
# Here is a single document with two sentences.
texts = [u"A storm hit the beach in Perth. It started to rain."]
for text in texts:
    sentences = nltk.sent_tokenize(text)
    for sentence in sentences:
      words = nltk.word_tokenize(sentence)
      tagged_words = nltk.pos_tag(words)
      ne_tagged_words = nltk.ne_chunk(tagged_words)
      print(ne_tagged_words)
```

```
(S A/DT storm/NN hit/VBD the/DT beach/NN in/IN (GPE Perth/NNP) ./.)
(S It/PRP started/VBD to/TO rain/VB ./.)
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
```

(continues on next page)

(continued from previous page)

```
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package maxent_ne_chunker is already up-to-date!
[nltk_data] Downloading package words to
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
```

3.2.2 Visualising NER in spaCy

We can use the `Doc.user_data` *attribute* to set a title for the visualisation.

```
from spacy import displacy
doc.user_data['title'] = "An example of an entity visualization"
displacy.render(doc, style='ent')
```

```
<IPython.core.display.HTML object>
```

3.2.3 Write the visualisation to a file

We can inform the render to not display the visualisation in the Jupyter Notebook instead write into a file by calling the render with two extra argument:

```
jupyter=False, page=True
```

```
from pathlib import Path
# the page=True indicates that we want to write to a file
html = displacy.render(doc, style='ent', jupyter=False, page=True)
output_path = Path("C:\\Users\\wei\\CITS4012\\ent_visual.html")
output_path.open("w", encoding="utf-8").write(html)
```

```
758
```

3.2.4 NLP pipeline in spaCy

Recall that spaCy's container objects represent linguistic units, suchas a text (i.e. document), a sentence and an individual token withlinguistic features already extracted for them.

How does spaCy create these containers and fill them withrelevant data?

A spaCy pipeline include, by default, a part-of-speech tagger (`tagger`), a dependency parser (`parser`), a lemmatizer (`lemmatizer`), an entity recognizer (`ner`), an attribute ruler (`attribute_ruler` and a word vectorisation model (`tok2vec`)).

```
import spacy
nlp = spacy.load('en_core_web_sm')
nlp.pipe_names
```

```
['tok2vec', 'tagger', 'parser', 'ner', 'attribute_ruler', 'lemmatizer']
```

spaCy allows you to load a selected set of pipeline components, dis-abling those that aren't necessary.

You can do this either when creating a nlp object or disable it after the nlp object is created.

Disabling when create

```
nlp = spacy.load('en_core_web_sm',disable=['parser'])
```

Disabling after creation

```
nlp.disable_pipes('tagger')
nlp.disable_pipes('parser')
```

3.2.5 Customising a NLP pipe in spaCy

```
import spacy
nlp = spacy.load('en_core_web_sm')

doc = nlp(u'I need a taxi to Cottesloe.')
for ent in doc.ents:
    print(ent.text, ent.label_)
```

```
Cottesloe GPE
```

What if?

If we would like to introduce a new entity type `SUBURB` for `Cottesloe` and other suburb names, how should we informthe NER component about it?

Steps of Customising a spaCy NER pipe

1. Create a training example to show the entity recognizer so it will learn what to apply the SUBURB label to;
2. Add a new label called SUBURB to the list of supported entitytypes;
3. Disable other pipe to ensure that only the entity recogniser will beupdated during training;
4. Start training;
5. Test your new NER pipe;
6. Serialise the pipe to disk;
7. Load the customised NER

```
import spacy
nlp = spacy.load('en_core_web_sm')
```

```
# Specify new label and training data
LABEL = 'SUBURB'
TRAIN_DATA = [('I need a taxi to Cottesloe',
                { 'entities': [(17, 26, 'SUBURB')] }),
              ('I like red oranges', { 'entities': []})]
```

```
# Add new label to the ner pipe
ner = nlp.get_pipe('ner')
ner.add_label(LABEL)
```

```
1
```

```
# Train
optimizer = nlp.create_optimizer()
import random
from spacy.tokens import Doc
from spacy.training import Example
for i in range(25):
  random.shuffle(TRAIN_DATA)
  for text, annotations in TRAIN_DATA:
    doc = Doc(nlp.vocab, words=text.split(" "))
    # We need to create a training example object
    example = Example.from_dict(doc, annotations)
    nlp.update([example], sgd=optimizer)
```

```
# Test
doc = nlp(u'I need a taxi to Crawley')
for ent in doc.ents:
  print(ent.text, ent.label_)
```

```
Crawley SUBURB
```

```
# Serialize the entire model to disk
nlp.to_disk('C:\\Users\\wei\\CITS4012') # Windows Path
```

```
# Load spacy model from disk
import spacy
nlp_updated = spacy.load('C:\\Users\\wei\\CITS4012')
```

```
# Test
doc = nlp_updated(u'I need a taxi to Subiaco')
for ent in doc.ents:
  print(ent.text, ent.label_)
```

```
Subiaco SUBURB
```

Your Turn

- Replace the suburb name with a few others, for example 'Claremont', 'Western Australia' and see what the the entity label is.
- Take a look at the directory and see how the `nlp` model is stored.
- This blog post on How to Train spaCy to Autodetect New Entities (NER) [Complete Guide] has more extensive examples on how to train a ner model with more data.

3.3 Finding Patterns

Imagine you are building a chat bot and we are trying to find utterances in user input that express one of the following:

What's Expressed

ability, possibility, permission, or obligation (as opposed to utterances that describe real actions that have occurred, are occurring, or occur regularly)

Example Sentences

For instance, we want to find "I can do it." but not "I've done it."

Linguistic Pattern

`subject + auxiliary + verb + . . . + direct object + ...`

The ellipses indicate that the direct object isn't necessarily located immediately behind the verb, there might be other words in between.

3.3.1 Check spaCy version

```
!pip show spacy
```

```
Name: spacy
Version: 3.0.5
Summary: Industrial-strength Natural Language Processing (NLP) in Python
Home-page: https://spacy.io
Author: Explosion
Author-email: contact@explosion.ai
License: MIT
Location: c:\programdata\anaconda3\envs\lda\lib\site-packages
Requires: cymem, blis, spacy-legacy, numpy, preshed, srsly, murmurhash, jinja2, thinc,
 ↪ tqdm, wasabi, requests, pydantic, setuptools, typer, catalogue, packaging, pathy
Required-by: en-core-web-sm
```

3.3.2 Hard-coded pattern discovery

To look for the `subject + auxiliary + verb + . . . + direct object + ...` pattern programmably, we need to go through each token's dependency label (*not part of speech label*) to first find the sequence of `nsubj aux ROOT` where `ROOT` indicate the root verb, then for each of children of the root verb (`ROOT`) we check to see if it is a direct object (`dobj`) of the verb.

```
import spacy
nlp = spacy.load('en_core_web_sm')
def dep_pattern(doc):
    for i in range(len(doc)-1):
        if doc[i].dep_ == 'nsubj' and doc[i+1].dep_ == 'aux' and doc[i+2].dep_ ==
↪'ROOT':
            for tok in doc[i+2].children:
                if tok.dep_ == 'dobj':
                    return True

    return False
```

```
# doc = nlp(u'We can overtake them.')
doc = nlp(u'I might send them a card as a reminder.')
```

Use displaycy to visualise the dependency

```
from spacy import displacy
displacy.render(doc, style='dep')
```

```
<IPython.core.display.HTML object>
```

```
options = {'compact': True, 'font': 'Tahoma'}
displacy.render(doc, style='dep', options=options)
```

```
<IPython.core.display.HTML object>
```

```
if dep_pattern(doc):
    print('Found')
else:
    print('Not found')
```

```
Found
```

Code Explanation

The `dep_pattern` function above takes a Doc object as parameter and returns a binary value `True` if the hard-coded pattern `subject + auxiliary + verb + . . . + direct object + ...` is found, otherwise `False`. The function iterates over the Doc object's tokens, searching for a `subject + auxiliary + verb`, where the `verb` is the `root` of the dependency tree. If the pattern is found, then we check whether the verb has a direct object among its syntactic children. Finally, if we find a direct object, the function returns `True`. Otherwise, it returns `False`.

3.3.3 Using spaCy pattern matcher

spaCy has a predefined tool called `Matcher`, that is specially designed to find sequences of tokens based on pattern rules. An implementation of the "subject + auxiliary + verb" pattern with `Matcher` might look like this:

```
import spacy
from spacy.matcher import Matcher
nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)
pattern = [{"DEP": "nsubj"}, {"DEP": "aux"}, {"DEP": "ROOT"}]
matcher.add("NsubjAuxRoot", [pattern])
doc = nlp("We can overtake them.")
matches = matcher(doc)
for match_id, start, end in matches:
    span = doc[start:end]
    print("Span: ", span.text)
    print("The positions in the doc are: ", start, "-", end)
    print("Match ID ", match_id)
    print(doc.vocab.strings[match_id])
    for tok in doc[end-1].children:
        if tok.dep_ == 'dobj':
            print("The direct object of {} is {}".format(doc[end-1], tok.dep_))
```

```
Span:  We can overtake
The positions in the doc are:  0 - 3
Match ID  10599197345289971701
NsubjAuxRoot
The direct object of overtake is dobj
```

Code Explanation

spaCy `Matcher` class takes a model's vocabulary as input and creates a matcher object named `matcher`. Then we need to define a pattern of interest. The pattern is specified in a dictionary object, and the order of the key value pairs indicate the desired sequence we are trying to find a match for. Once the pattern is found, a list of tuples in the form of `(match_id, start, end)` is returned. The `match_id` is the hash value of the string ID "NsubjAuxRoot". To get the string value, you can look up the ID in the StringStore.

3.3.4 Summary of Rule-based Matching

Steps for using the Matcher class:

1. Create a Matcher instance by passing in a shared Vocab object;
2. Specify the pattern as an list of dependency labels;
3. Add the pattern to the a Matcher object;
4. Input a Doc object to the matcher;
5. Go through each match $\langle match\_id, start, end \rangle$.

We have seen a *Dependency Matcher* just now, there are more Rule-based matching support in spaCy:

- Token Matcher: regex, and patterns such as

- Phrase Matcher: PhraseMatcher class
- Entity Ruler
- Combining models with rules

For more information of different types of matchers, see spaCy Documentation on Rule Based Matching.

Reference: Chapter 6 of NATURAL LANGUAGE PROCESSING WITH PYTHON AND SPACY

3.4 Your first chatbot

A typical chatbot app consists of multiple tiers. After you've implemented the logic for processing user input on your machine, you'll need a messenger app (e.g. Skype, Facebook Messenger or Telegram) that allows you to create accounts that your programs operate. Users won't interact with the bot implementation on your machine directly; instead, they'll chat with the bot through the messenger API. Apart from a messenger, your chatbot might require some additional services, such as a database or other storage.

3.4.1 Creating a Telegram Account and Authorizing Your Bot

Note: You'll need a smartphone or tablet that runs either iOS or Android to create a Telegram account. A PC version of Telegram won't work for this operation. However, once you create a Telegram account, you can use it on a PC.

On your smartphone or tablet go to an app store, and install the `Telegram app`. Then you need to enter your `phone number` to access the app.

1. In the Telegram app, perform a search for @BotFather (note: case sensitive) or open the URL https://telegram.me/botfather/. BotFather is a Telegram bot that manages all the other bots in your account.
2. On the BotFather page, click the Start button to see the list of commands that you can use to set up your Telegram bots.
3. To create a new bot, enter the `/newbot` command in the Write a message box. You'll be prompted for a name and a username for your bot. Then you'll be given an authorization token for the new bot. Note down the token in a secure place as people who have access to the token can manipulate your new bot.

3.4.2 Install the `python-telegram-bot` Library

To connect chatbot functionality implemented in Python, you'll need the python-telegram-bot library, which is built on top of the Telegram Bot API. The library provides an easy-to-use interface for bot programmers developing apps for Telegram.

```
pip install python-telegram-bot --upgrade
```

Once you've installed the library, use the following lines of code to perform a quick test to verify that you can access your Telegram bot from Python. You must have an internet connection for this test to work.

```
import telegram
TOKEN = 'YOUR TOKEN'
bot = telegram.Bot(token=TOKEN)
```

```
print(bot.get_me())
```

```
{'first_name': 'Pizza2me', 'is_bot': True, 'supports_inline_queries': False, 'id':␣
↪1947674606, 'username': 'Pizza2meBot', 'can_join_groups': True, 'can_read_all_group_
↪messages': False}
```

3.4.3 A copy-cat chatbot

A chatbot simply echoing the input message.

```
from telegram.ext import Updater, MessageHandler, Filters
#function that implements the message handler
def echo(update, context):
    update.message.reply_text(update.message.text)
#creating an Updater instance
updater = Updater(TOKEN, use_context=True)
#registering a handler to handle input text messages
updater.dispatcher.add_handler(MessageHandler(Filters.text, echo))
#starting polling updates from the messenger
updater.start_polling()
updater.idle()
```

Click the `run` button for the above cell, it will start the bot. You can then go to your smartphone or tablet's Telegram app to chat with your bot.

Note: You can use the Interrupt button in your notebook to stop the bot after testing.

3.4.4 Chatbot that uses spaCy code

```
import spacy
from telegram.ext import Updater, MessageHandler, Filters
#the callback function that uses spaCy
def utterance(update, context):
    msg = update.message.text
    nlp = spacy.load('en_core_web_sm')
    doc = nlp(msg)
    for token in doc:
        if token.dep_ == 'dobj':
            update.message.reply_text('We are processing your request...')
            return
    update.message.reply_text('Please rephrase your request. Be as specific as␣
↪possible!')

#the code responsible for interactions with Telegram
updater = Updater(TOKEN, use_context=True)
updater.dispatcher.add_handler(MessageHandler(Filters.text, utterance))
updater.start_polling()
updater.idle()
```

Your Turn

Integrte some more patterns to handle more varieties of requests. For example, extracting the type of pizza by finding its left children, if you are handling pizza ordering.

Reference: Chapter 11 of NATURAL LANGUAGE PROCESSING WITH PYTHON AND SPACY

3.5 Exercise

Extend a `to + GPE` NER pattern

Let's consider a travel assistant here. Write a function that uses the entity type `GPE` to find the desired destination of a user. The code below is capable of very simple parsing, but not able to handle sentences like `I am going to a conference in Berlin.`

- Modify the code to make it work for more cases.
- Incoporate that into your Telgram booking bot.

```
import spacy
nlp = spacy.load('en_core_web_sm')
```

```
# Here's the function that figures out the destination
def det_destination(doc):
    for i, token in enumerate(doc):
        if token.ent_type != 0 and token.ent_type_ == 'GPE':
            while True:
                token = token.head
                if token.text == 'to':
                    return doc[i].text
                if token.head == token:
                    return 'Failed to determine'
    return 'Failed to determine'
```

```
# Testing the det_destination function
doc = nlp(u'I am going to Berlin.')
# doc = nlp(u'I am going to the conference in Berlin.')
for token in doc:
    print(token.text, token.ent_type_, token.head)
```

```
I  going
am  going
going  going
to  going
Berlin GPE to
.  going
```

```
dest = det_destination(doc)
print('It seems the user wants a ticket to ' + dest)
```

```
It seems the user wants a ticket to Berlin
```

CHAPTER

FOUR

LAB04: COUNT-BASED MODELS

In this lab, we will look at how to process natural language text to build two different types of count-based matrices, one for word characterisation (i.e. word co-occurence matrix), one for document characterisation (i.e. document term matrix).

While putting the theory into practice, we will also introduce two more packages that are excelled at count-based methods, namely `scikit learn` and `gensim`. After introducing the basics using small toy corpus, we demonstrate how `tf-idf` can be used for document classification in `scikit learn`.

Work on your project after you finish this lab.

4.1 TF-IDF in scikit-learn and Gensim

In a large text corpus, some words will be very present (e.g. "the", "a", "is" in English) hence carrying very little meaningful information about the actual contents of the document. If we were to feed the raw count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms. In other words, frequent words may not provide descriminative or similarity information for

- scoring/ranking documents with regard to a query document, in the context of information retrieval used in search engines;
- separating documents into different categories, in the context of documen classification (sentiment detection, spam detection).

In this lab, we will focus on document classification. In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the tf–idf transform.

TF-IDF was originally a term weighting scheme developed for information retrieval (as a ranking function for search engines results) that has also found good use in document classification and clustering.

Term Frequency

Denoted $tf_{d,t}$, which means term-frequency of term t in document d, can be referred to as the *raw count*, i.e. the number of times term term t occurs in document d; or the raw count normalised (divided) by the total number of words in document d.

Document Frequency

Denoted df_t, which means the number of times that term t occurs across the entire collection of documents (i.e. corpus). It is a value specific to each term but not specific to each document.

Inverse Document Frequency

Denoted idf_t, which is defined as

$$idf_t = \log_2 \frac{N}{df_t}$$

where N is total number of documents in the collection. It is a value specific to each term but not specific to each document.

TF-IDF is the term frequency discounted by the document freqency. In other words, a frequent term in a document needs to be infrequent across documents to acquire a high tf-idf value. There are several variations of TF-IDF implementations. Gensim's implementation is more closer to the original definition. scikit-learn's implementation normalises the resulting vector to ensure the values are between 0 and 1, which is better for classification tasks.

```
text = ["It was the best of times",
"it was the worst of times",
"it was the age of wisdom",
"it was the age of foolishness"]
```

4.1.1 TF-IDF in Gensim

Gensim is yet another popular libray specialising in statistical analysis of natural language text, in particular useful for Topic Modelling, which we will cover later in the unit.

Note: You will need to activate the cits4012_py37 environment to use gensim.

```
import gensim
import gensim.downloader as api
from gensim.models import TfidfModel
from gensim.corpora import Dictionary
```

Step 1: Preprocessing

Gensim has a different routine in preparing text. It uses `gensim.utils.simple_preprocess()` to tokenise while removing punctuation and turn the tokens into lower cases.

```
def sent_to_words(sentences):
    for sentence in sentences:
        # deacc=True removes punctuations
        yield(gensim.utils.simple_preprocess(str(sentence), deacc=True))
```

Step 2: Create a corpus with counts

Gensim has a built-in class `gensim.corpora.Dictionary` that has a function `doc2bow` that implements the bag of words idea, which processes the document collection, assigning an id to each unique token, while counting the term frequency of each token in each document. The following code returns each document as a list of tuples, in the form of `(term_id, count)`

```
doc_tokenized = list(sent_to_words(text))
dictionary = Dictionary()
BoW_corpus = [dictionary.doc2bow(doc, allow_update=True) for doc in doc_tokenized]
BoW_corpus
```

```
[[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1)],
 [(1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)],
 [(1, 1), (2, 1), (3, 1), (5, 1), (7, 1), (8, 1)],
 [(1, 1), (2, 1), (3, 1), (5, 1), (7, 1), (9, 1)]]
```

We can examine the Bag of Words corpus obtained. You can think of the dictionary object is a vocabulary of the collection, mapping a term id to its lexical form (i.e. the word form) of the term.

```
for doc in BoW_corpus:
   print([[dictionary[id], freq] for id, freq in doc])
```

```
[['best', 1], ['it', 1], ['of', 1], ['the', 1], ['times', 1], ['was', 1]]
[['it', 1], ['of', 1], ['the', 1], ['times', 1], ['was', 1], ['worst', 1]]
[['it', 1], ['of', 1], ['the', 1], ['was', 1], ['age', 1], ['wisdom', 1]]
[['it', 1], ['of', 1], ['the', 1], ['was', 1], ['age', 1], ['foolishness', 1]]
```

Step 3: Calculating the tfidf values

A `gensim.models.TfidfModel` object can be constructed using the processed BoW corpus. The `smartirs` parameter stands for SMART information retrieval system, where SMART is an acronym for "System for the Mechanical Analysis and Retrieval of Text". If interested, you can read more about SMART on Wikipedia, which contains a rather comprehensive list of TF-IDF variants. `smartirs = ntc` means the model will use `n` raw term freqency, `t` zero-corrected idf, and `c` cosine for document vector nomalisation. For a list of other letter codes for each of the three tf-idf components, see the tabs or refer to the original documentation. You do not need to memorise all these variants, but just be aware of the many alternatives in calculating such a seemingly simple value. You can even define your own way of calculating tf and idf to feed into the model constructor.

Term frequency weighing

- b - binary,
- t or n - raw,
- a - augmented,
- l - logarithm,
- d - double logarithm,
- L - log average.

Document frequency weighting

- x or n - none,
- f - idf,
- t - zero-corrected idf,
- p - probabilistic idf.

Document normalization

- x or n - none,
- c - cosine,
- u - pivoted unique,
- b - pivoted character length.

Example Code

```
import numpy as np
tfidf = TfidfModel(BoW_corpus, smartirs='ntc')

# Get the tfidf vector representation of the second sentence
tfidf[BoW_corpus[1]]
```

```
[(1, 0.11713529839512132),
 (2, 0.11713529839512132),
 (3, 0.11713529839512132),
 (4, 0.48099076877929253),
 (5, 0.11713529839512132),
 (6, 0.8448462391634637)]
```

```
# Get the tfidf transformed corpus,
# then the vector of the second sentence.
tfidf[BoW_corpus][1]
```

```
[(1, 0.11713529839512132),
 (2, 0.11713529839512132),
 (3, 0.11713529839512132),
 (4, 0.48099076877929253),
 (5, 0.11713529839512132),
 (6, 0.8448462391634637)]
```

```
# Now a friendlier print out
for doc in tfidf[BoW_corpus]:
   print([[dictionary[id], np.around(freq,decimals=2)] for id, freq in doc])
```

```
[['best', 0.84], ['it', 0.12], ['of', 0.12], ['the', 0.12], ['times', 0.48], ['was',␣
 ↪0.12]]
[['it', 0.12], ['of', 0.12], ['the', 0.12], ['times', 0.48], ['was', 0.12], ['worst',␣
 ↪0.84]]
```

(continues on next page)

(continued from previous page)

```
[['it', 0.12], ['of', 0.12], ['the', 0.12], ['was', 0.12], ['age', 0.48], ['wisdom',␣
↪0.84]]
[['it', 0.12], ['of', 0.12], ['the', 0.12], ['was', 0.12], ['age', 0.48], [
↪'foolishness', 0.84]]
```

How do we get the document-term matrix

```
vocab = [dictionary[i] for i in range(len(dictionary))]
vocab
```

```
['best',
 'it',
 'of',
 'the',
 'times',
 'was',
 'worst',
 'age',
 'wisdom',
 'foolishness']
```

```
index = list(range(len(BoW_corpus)))
index
```

```
[0, 1, 2, 3]
```

```
import pandas as pd
df = pd.DataFrame(data=np.zeros((len(BoW_corpus), len(vocab)), dtype=np.float16),
                  index=index,
                  columns=vocab)
```

```
for idx in index:
    for id, freq in tfidf[BoW_corpus[idx]]:
        df[dictionary[id]][idx] = freq
```

```
df
```

```
       best        it        of       the     times       was     worst  \
0  0.844727  0.117126  0.117126  0.117126  0.480957  0.117126  0.844727
1  5.000000  0.117126  0.117126  0.117126  0.480957  0.117126  0.844727
2  0.000000  0.117126  0.117126  0.117126  0.000000  0.117126  0.000000
3  0.000000  0.117126  0.117126  0.117126  0.000000  0.117126  0.000000

        age    wisdom  foolishness
0  0.480957  0.844727     0.844727
1  0.000000  0.000000     0.000000
2  0.480957  0.844727     0.000000
3  0.480957  0.000000     0.844727
```

4.1.2 TF-IDF in scikit-learn

In scikit-learn, the TF-IDF is caculated using the `TfidfTransformer`. Its default settings, `TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)` the term frequency, the number of times a term occurs in a given document, is multiplied with idf component, which is computed as

TF-IDF with `smooth_idf=True`

$$idf_t = \log_2 \frac{1+n}{1+df_t} + 1$$

The default parameter `smooth_idf=True` adds "1" to the numerator and denominator as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions.

TF-IDF with `smooth_idf=False`

$$idf_t = \log_2 \frac{n}{df_t} + 1$$

With `smooth_idf=False`, the "1" count is added to the idf instead of the idf's denominator:

Then the resulting tf-idf vectors are then normalized by the Euclidean norm

$$v_{norm} = \frac{v}{||v||} = \frac{v}{\sqrt{v_1^2 + ... + v_n^2}}$$

See scikit-learn Documenation on Feature Extraction (Section 6.2.3.4) for more details.

`TfidfTransformer` vs `TfidfVectorizer`

With `TfidfTransformer` you will systematically compute word counts using `CountVectorizer` and then compute the Inverse Document Frequency (IDF) values and only then compute the Tf-idf scores.

With `TfidfVectorizer` on the contrary, you will do all three steps at once. Under the hood, it computes the word counts, IDF values, and Tf-idf scores all using the same dataset.

Use `TfidfTransformer` if you need to obtain term frequency (i.e. term counts). We will illustrate the TF-IDF calculation using `TfidfVectorizer` in this notebook and `TfidfTransformer` in the classification notebook after this.

Example Code

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(text).toarray()
tfidf
```

```
array([[0.        , 0.60735961, 0.        , 0.31694544, 0.31694544,
        0.31694544, 0.4788493 , 0.31694544, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.31694544, 0.31694544,
        0.31694544, 0.4788493 , 0.31694544, 0.        , 0.60735961],
```

(continues on next page)

(continued from previous page)

```
       [0.4788493 , 0.        , 0.        , 0.31694544, 0.31694544,
        0.31694544, 0.        , 0.31694544, 0.60735961, 0.        ],
       [0.4788493 , 0.        , 0.60735961, 0.31694544, 0.31694544,
        0.31694544, 0.        , 0.31694544, 0.        , 0.        ]])
```

```
import numpy as np
# Define labels for the x and y axis
nrows = np.shape(tfidf)[0]
xlabels = tfidf_vectorizer.get_feature_names()
ylabels = ['D' + str(idx) for idx in list(np.arange(nrows))]
```

```
['D0', 'D1', 'D2', 'D3']
```

```
xlabels
```

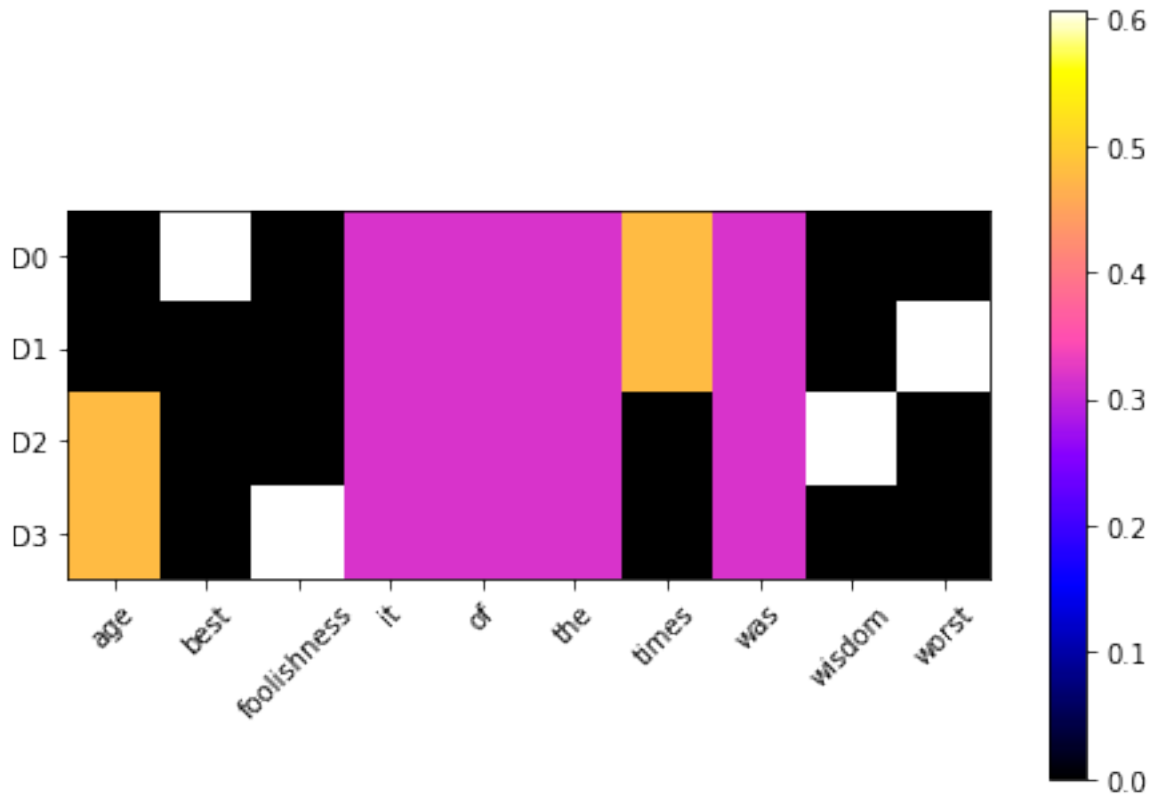
```
['age',
 'best',
 'foolishness',
 'it',
 'of',
 'the',
 'times',
 'was',
 'wisdom',
 'worst']
```

```
ylabels
```

```
['D0', 'D1', 'D2', 'D3']
```

```
import matplotlib.pyplot as plt

plt.figure()
plt.axes([0, 0, 1, 1])
plt.imshow(tfidf, interpolation='nearest',
          cmap=plt.cm.gnuplot2, vmin=0)
plt.xticks(range(len(xlabels)), xlabels, rotation=45)
plt.yticks(range(nrows), ylabels)
plt.colorbar()
plt.tight_layout()
plt.show()
```

```
<ipython-input-10-4c5527a74677>:10: UserWarning: This figure includes Axes that are␣
 ↪not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
```



Note, this is a document-term matrix, with tf-idf values for each cell, not a similarity or distance plot.

Your Turn

You can refer to the code for similarity calculation in Resturant Example (Lecture 3) to work out the cosine similarities between these documents. Note this will require `pytorch`.

4.2 Document Classification

Now that we have a good understanding of TF-IDF term document matrix, we can treat each term as a feature, and each document (row) as an instance or a training sample to train a classifier. The classifier can be any traditional supervised learning models that deals with tabular shaped data, where one column stores the labels of each sample. All other columns are feature variables, in this case, each term/word is a feature. An illustration table is shown below.

ID	best	it	of	the	times	was	worst	age	wis-dom	foolish-ness	class label
0	0.844727	0.117126	0.117126	0.117126	0.480957	0.117126	0.844727	0.480957	0.844727	0.844727	positive
1	5.000000	0.117126	0.117126	0.117126	0.480957	0.117126	0.844727	0.000000	0.000000	0.000000	negative
2	0.000000	0.117126	0.117126	0.117126	0.000000	0.117126	0.000000	0.480957	0.844727	0.000000	positive
3	0.000000	0.117126	0.117126	0.117126	0.000000	0.117126	0.000000	0.480957	0.000000	0.844727	negative

The goal of this guide is to explore some of the main 'scikit-learn' tools on a popular classification task: analyzing a collection of text documents (newsgroups posts) and classify them into one of the twenty different topics.

In this notebook we will see how to:

- load the file contents and the categories
- extract feature vectors suitable for machine learning
- train a linear model to perform categorization
- use a grid search strategy to find a good configuration of both the feature extraction components and the classifier

Original Notebook Credit to scikit-learn tutorial on Working with Text Data

4.2.1 Loading the 20 newsgroups dataset

The dataset is called "Twenty Newsgroups". Here is the official description:

> The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. To the best of our knowledge, it was originally collected by Ken Lang, probably for his paper "Newsweeder: Learning to filter netnews," though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering.

In the following we will use the built-in dataset loader for 20 newsgroups from scikit-learn.

In order to get faster execution times for this first example we will work on a partial dataset with only 4 categories out of the 20 available in the dataset:

```
categories = ['alt.atheism', 'soc.religion.christian',
              'comp.graphics', 'sci.med']
```

We can now load the list of files matching those categories as follows (this may take a while - 65.1s on a desktop computer - AMD 16 core, 32GB RAM):

```
from sklearn.datasets import fetch_20newsgroups
twenty_train = fetch_20newsgroups(subset='train',
      categories=categories, shuffle=True, random_state=42)
```

The returned dataset is a `scikit-learn` "bunch": a simple holder object with fields that can be both accessed as python `dict` keys or `object` attributes for convenience, for instance the `target_names` holds the list of the requested category names::

```
twenty_train.target_names
```

```
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
```

The files themselves are loaded in memory in the `data` attribute. For reference the filenames are also available:

```
len(twenty_train.data)
```

```
2257
```

```
twenty_train.filenames[0]
```

```
'C:\\Users\\wei\\scikit_learn_data\\20news_home\\20news-bydate-train\\comp.graphics\\
↪38440'
```

Let's print the first three lines of the first loaded file:

```
print("\n".join(twenty_train.data[0].split("\n")[:3]))
```

```
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
Nntp-Posting-Host: hampton
```

Below is how to access the class label (i.e. target column) of the first document.

```
print(twenty_train.target_names[twenty_train.target[0]])
```

```
comp.graphics
```

Supervised learning algorithms will require a category label for each document in the training set. In this case the category is the name of the newsgroup which also happens to be the name of the folder holding the individual documents.

For speed and space efficiency reasons `scikit-learn` loads the target attribute as an array of integers that corresponds to the index of the category name in the `target_names` list. The category integer id of each sample is stored in the `target` attribute:

```
twenty_train.target[:10]
```

```
array([1, 1, 3, 3, 3, 3, 3, 2, 2, 2], dtype=int64)
```

It is possible to get back the category names as follows:

```
for t in twenty_train.target[:10]:
    print(twenty_train.target_names[t])
```

```
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
sci.med
sci.med
sci.med
```

You might have noticed that the samples were shuffled randomly when we called `fetch_20newsgroups(..., shuffle=True, random_state=42)`: this is useful if you wish to select only a subset of samples to quickly train a model and get a first idea of the results before re-training on the complete dataset later.

4.2.2 Extracting features from text files

In order to perform machine learning on text documents, we first need to turn the text content into numerical feature vectors.

Bags of words

The most intuitive way to do so is to use a bags of words representation:

1. Assign a fixed integer id to each word occurring in any document of the training set (for instance by building a dictionary from words to integer indices).
2. For each document `#i`, count the number of occurrences of each word `w` and store it in `X[i, j]` as the value of feature `#j` where `j` is the index of word `w` in the dictionary.

The bags of words representation implies that `n_features` is the number of distinct words in the corpus: this number is typically larger than 100,000.

If `n_samples == 10000`, storing `X` as a NumPy array of type float32 would require 10000 x 100000 x 4 bytes = **4GB in RAM** which is barely manageable on today's computers.

Fortunately, **most values in X will be zeros** since for a given document less than a few thousand distinct words will be used. For this reason we say that bags of words are typically **high-dimensional sparse datasets**. We can save a lot of memory by only storing the non-zero parts of the feature vectors in memory.

`scipy.sparse` matrices are data structures that do exactly this, and `scikit-learn` has built-in support for these structures.

Tokenizing text with `scikit-learn`

Text preprocessing, tokenizing and filtering of stopwords are all included in class: `CountVectorizer`, which builds a dictionary of features and transforms documents to feature vectors:

```
from sklearn.feature_extraction.text import CountVectorizer

count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(twenty_train.data)
X_train_counts.shape
```

```
(2257, 35788)
```

Class `CountVectorizer` supports counts of N-grams of words or consecutive characters. Once fitted, the vectorizer has built a dictionary of feature indices:

```
# The number of times the word 'algorithm' occurs
count_vect.vocabulary_.get(u'algorithm')
```

```
4690
```

The index value of a word in the vocabulary is linked to its frequency in the whole training corpus.

Note: The method `count_vect.fit_transform` performs two actions: it learns the vocabulary and transforms the documents into count vectors. It's possible to separate these steps by calling `count_vect.fit(twenty_train.data)` followed by `X_train_counts = count_vect.transform(twenty_train.data)`, but doing so would tokenize and vectorize each text file twice.

From occurrences to frequencies

Occurrence count is a good start but there is an issue: longer documents will have higher average count values than shorter documents, even though they might talk about the same topics.

To avoid these potential discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called `tf` for Term Frequencies.

Another refinement on top of tf is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus.

This downscaling is the `tf-idf` - "Term Frequency times Inverse Document Frequency" we discussed earlier.

As dicussed in the previous notebooks, both **tf** and **tf–idf** can be computed as follows using class `TfidfTransformer`:

```
from sklearn.feature_extraction.text import TfidfTransformer

tf_transformer = TfidfTransformer(use_idf=False).fit(X_train_counts)
X_train_tf = tf_transformer.transform(X_train_counts)
X_train_tf.shape
```

```
(2257, 35788)
```

In the above example-code, we firstly use the `fit(..)` method to fit our estimator to the data and secondly the `transform(..)` method to transform our count-matrix to a tf-idf representation. These two steps can be combined to achieve the same end result faster by skipping redundant processing. This is done through using the `fit_transform(..)` method as shown below, and as mentioned in the note in the previous section:

```
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
X_train_tfidf.shape
```

```
(2257, 35788)
```

```
X_train_tfidf.toarray()
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

4.2.3 Training a classifier

Now that we have our features, we can train a classifier to try to predict the category of a post. Let's start with a `naïve Bayes <naive_bayes>` classifier, which provides a nice baseline for this task. `scikit-learn` includes several variants of this classifier; the one most suitable for word counts is the multinomial variant:

```
from sklearn.naive_bayes import MultinomialNB

clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target)
```

To try to predict the outcome on a new document we need to extract the features using almost the same feature extracting chain as before. The difference is that we call `transform` instead of `fit_transform` on the transformers, since they have already been fit to the training set:

```
docs_new = ['God is love', 'OpenGL on the GPU is fast']
X_new_counts = count_vect.transform(docs_new)
X_new_tfidf = tfidf_transformer.transform(X_new_counts)

predicted = clf.predict(X_new_tfidf)

for doc, category in zip(docs_new, predicted):
  print('%r => %s' % (doc, twenty_train.target_names[category]))
```

```
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

4.2.4 Building a pipeline

In order to make the vectorizer => transformer => classifier easier to work with, `scikit-learn` provides a class `~sklearn.pipeline.Pipeline` that behaves like a compound classifier:

```
from sklearn.pipeline import Pipeline
text_clf = Pipeline([
     ('vect', CountVectorizer()),
     ('tfidf', TfidfTransformer()),
     ('clf', MultinomialNB())])
```

The names `vect`, `tfidf` and `clf` (classifier) are arbitrary. We will use them to perform grid search for suitable hyperparameters below. We can now train the model with a single command:

```
text_clf.fit(twenty_train.data, twenty_train.target)
```

```
Pipeline(steps=[('vect', CountVectorizer()), ('tfidf', TfidfTransformer()),
                ('clf', MultinomialNB())])
```

4.2.5 Evaluation of the performance on the test set

Evaluating the predictive accuracy of the model is simply a comparision of the predicted and the actual labels.

```
import numpy as np
twenty_test = fetch_20newsgroups(subset='test',
              categories=categories, shuffle=True, random_state=42)
docs_test = twenty_test.data
predicted = text_clf.predict(docs_test)
np.mean(predicted == twenty_test.target)
```

```
0.8348868175765646
```

We achieved 83.5% accuracy. Let's see if we can do better with a linear `support vector machine (SVM) <svm>`, which is widely regarded as one of the best text classification algorithms (although it's also a bit slower than naïve Bayes). We can change the learner by simply plugging a different classifier object into our pipeline:

```
from sklearn.linear_model import SGDClassifier
text_clf = Pipeline([
     ('vect', CountVectorizer()),
     ('tfidf', TfidfTransformer()),
     ('clf', SGDClassifier(loss='hinge', penalty='l2',
                           alpha=1e-3, random_state=42,
                           max_iter=5, tol=None))])

text_clf.fit(twenty_train.data, twenty_train.target)
```

```
Pipeline(steps=[('vect', CountVectorizer()), ('tfidf', TfidfTransformer()),
                ('clf',
                 SGDClassifier(alpha=0.001, max_iter=5, random_state=42,
                               tol=None))])
```

```
predicted = text_clf.predict(docs_test)
np.mean(predicted == twenty_test.target)
```

```
0.9101198402130493
```

We achieved 91.3% accuracy using the SVM. `scikit-learn` provides further utilities for more detailed performance analysis of the results:

```
from sklearn import metrics
print(metrics.classification_report(twenty_test.target, predicted,
      target_names=twenty_test.target_names))
```

```
                        precision    recall  f1-score   support

           alt.atheism       0.95      0.80      0.87       319
         comp.graphics       0.87      0.98      0.92       389
               sci.med       0.94      0.89      0.91       396
soc.religion.christian       0.90      0.95      0.93       398

              accuracy                           0.91      1502
             macro avg       0.91      0.91      0.91      1502
          weighted avg       0.91      0.91      0.91      1502
```

```
metrics.confusion_matrix(twenty_test.target, predicted)
```

```
array([[256,  11,  16,  36],
       [  4, 380,   3,   2],
       [  5,  35, 353,   3],
       [  5,  11,   4, 378]], dtype=int64)
```

As expected the confusion matrix shows that posts from the newsgroups on atheism and Christianity are more often confused for one another than with computer graphics.

Note: SGD stands for Stochastic Gradient Descent. This is a simple optimization algorithms that is known to be scalable when the dataset has many samples.

By setting `loss="hinge"` and `penalty="l2"` we are configuring the classifier model to tune its parameters for the linear Support Vector Machine cost function.

Alternatively we could have used `sklearn.svm.LinearSVC` (Linear Support Vector Machine Classifier) that provides an alternative optimizer for the same cost function based on the liblinear_ C++ library.

4.2.6 Parameter tuning using grid search

We've already encountered some parameters such as `use_idf` in the `TfidfTransformer`. Classifiers tend to have many parameters as well; e.g., `MultinomialNB` includes a smoothing parameter `alpha` and `SGDClassifier` has a penalty parameter `alpha` and configurable loss and penalty terms in the objective function (see the module documentation, or use the Python `help` function to get a description of these).

Instead of tweaking the parameters of the various components of the chain, it is possible to run an exhaustive search of the best parameters on a grid of possible values. We try out all classifiers on either words or bigrams, with or without idf, and with a penalty parameter of either 0.01 or 0.001 for the linear SVM:

```
from sklearn.model_selection import GridSearchCV
parameters = {
    'vect__ngram_range': [(1, 1), (1, 2)],
    'tfidf__use_idf': (True, False),
    'clf__alpha': (1e-2, 1e-3),
}
```

Obviously, such an exhaustive search can be expensive. If we have multiple CPU cores at our disposal, we can tell the grid searcher to try these eight parameter combinations in parallel with the `n_jobs` parameter. If we give this parameter a value of `-1`, grid search will detect how many cores are installed and use them all:

```
gs_clf = GridSearchCV(text_clf, parameters, cv=5, n_jobs=-1)
```

The grid search instance behaves like a normal `scikit-learn` model. Let's perform the search on a smaller subset of the training data to speed up the computation:

```
gs_clf = gs_clf.fit(twenty_train.data[:400], twenty_train.target[:400])
```

After calling `fit` on a `GridSearchCV` object, we now obtained a classifier that we can use to `predict`:

```
twenty_train.target_names[gs_clf.predict(['God is love'])[0]]
```

```
'soc.religion.christian'
```

The object's `best_score_` and `best_params_` attributes store the best mean score and the parameters setting corresponding to that score:

```
gs_clf.best_score_
```

```
0.9175000000000001
```

```
for param_name in sorted(parameters.keys()):
    print("%s: %r" % (param_name, gs_clf.best_params_[param_name]))
```

```
clf__alpha: 0.001
tfidf__use_idf: True
vect__ngram_range: (1, 1)
```

A more detailed summary of the search is available at `gs_clf.cv_results_`.

The `cv_results_` parameter can be easily imported into pandas as a `DataFrame` for further inspection.

Note: A `GridSearchCV` object also stores the best classifier that it trained as its `best_estimator_` attribute. In this case, that isn't much use as we trained on a small, 400-document subset of our full training set.

The index value of a word in the vocabulary is linked to its frequency in the whole training corpus.

Your Turn

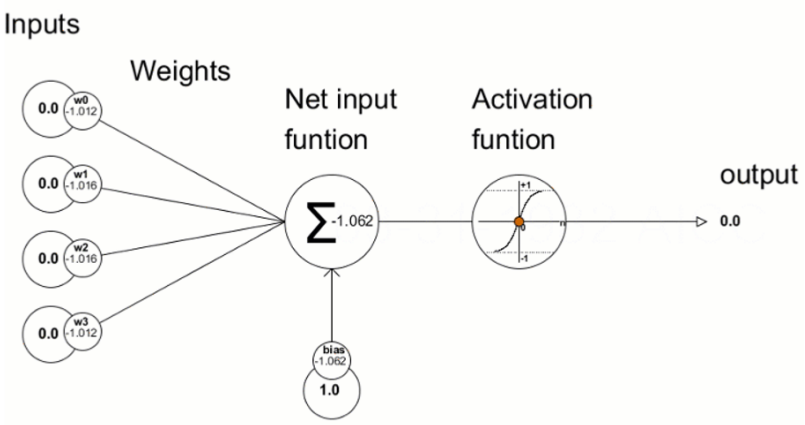
1. Build classifiers using the `raw count`, or `term frequency` to compare the performance against `tf-idf`.
2. Update the code to use TfidfVectorisor.
3. Think about how a classification model or a similarity/distance calculation using `tf-idf` might help your chatbot.

CHAPTER

FIVE

LAB05: INTRODUCTION TO NEURAL NETWORKS AND PYTORCH

In this lab, we will introduce the basic training steps of neural networks by starting with simple linear models in Numpy. Then we will introduce basic Pytorch data types, and re-implement the linear model in Pytorch. A number of important Pytorch functions are introduced in this lab.



The Linear Models in Numpy/Pytorch notebooks are adapted from: Deep Learning in Pytorch Step by Step Chapter 1 notebook on Github

5.1 Linear Models in Numpy

Despite the simplicity, linear models can be applicable in many scenarios. For example, a traditional model for document sentiment classification can be achieved by collecting a set of features (e.g. fequency of positive and negative words by looking up SentiWordNet, and model the sentiment score as a linear combination of the features. Linear models can be seen as the simplest type of neural networks without non-linear activation functions.

Notebook adapted from Github of Deep Learning with Pytorch: Step by Step

```
import numpy as np
import matplotlib.pyplot as plt
```

5.1.1 Data Preparation

Generating a synthetic dataset

We define a linear line $y = 2x + 1$ and add random small guassian noise for the synthetic dataset.

```
true_b = 1
true_w = 2
N = 100
# Data Generation
np.random.seed(42)
x = np.random.rand(N, 1)
# Guassian noise to add some randomness to y
epsilon = (.1 * np.random.randn(N, 1))
y = true_b + true_w * x + epsilon
```

```
print("The epsilon is in the range of [%3.2f, %3.2f]" % (min(epsilon), max(epsilon)))
```

```
The epsilon is in the range of [-0.20, 0.25]
```

Splitting the dataset to training and validation

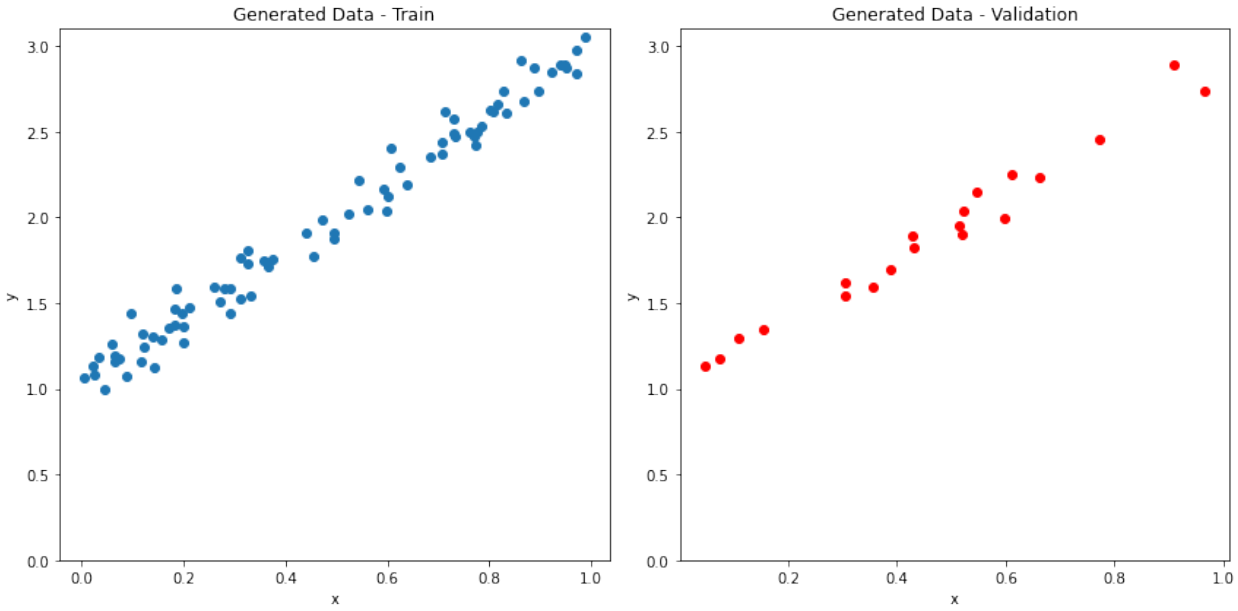
Next, let's split our synthetic data into train and validation sets, shuffling the array of indexes and using the first 80 shuffled points for training, the rest of 20 for validation.

```
# Shuffles the indices
idx = np.arange(N)
np.random.shuffle(idx)
# Uses first 80 random indices for train
train_idx = idx[:int(N*.8)]
# Uses the remaining indices for validation
val_idx = idx[int(N*.8):]
# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

```
def figure1(x_train, y_train, x_val, y_val):
    fig, ax = plt.subplots(1, 2, figsize=(12, 6))

    ax[0].scatter(x_train, y_train)
    ax[0].set_xlabel('x')
    ax[0].set_ylabel('y')
    ax[0].set_ylim([0, 3.1])
    ax[0].set_title('Generated Data - Train')

    ax[1].scatter(x_val, y_val, c='r')
    ax[1].set_xlabel('x')
    ax[1].set_ylabel('y')
    ax[1].set_ylim([0, 3.1])
    ax[1].set_title('Generated Data - Validation')
    fig.tight_layout()

    return fig, ax
```

```
figure1(x_train, y_train, x_val, y_val)
```

```
(<Figure size 864x432 with 2 Axes>,
 array([<AxesSubplot:title={'center':'Generated Data - Train'}, xlabel='x', ylabel='y
↪'>,
        <AxesSubplot:title={'center':'Generated Data - Validation'}, xlabel='x',␣
↪ylabel='y'>],
       dtype=object))
```



5.1.2 Gradient Descent

Step 0: Random Initialization

For training a model, you need to randomly initialize the parameters/weights (we have only two, b and w).

```
# Step 0 - Initializes parameters "b" and "w" randomly
np.random.seed(42)
b = np.random.randn(1)
w = np.random.randn(1)
print(b, w)
```

```
[0.49671415] [-0.1382643]
```

Step 1: Compute Predication

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train
yhat
```

```
array([[0.390075  ],
       [0.4879263 ],
       [0.37737776],
       [0.4931996 ],
       [0.39550552],
       [0.48647642],
       [0.46923887],
       [0.4537164 ],
       [0.43142369],
       [0.36265521],
       [0.47984062],
       [0.38954095],
       [0.38579895],
       [0.39583566],
       [0.48320959],
       [0.47115538],
       [0.47514234],
       [0.49386805],
       [0.36026292],
       [0.41053427],
       [0.39881299],
       [0.41404593],
       [0.36925186],
       [0.40856271],
       [0.45787094],
       [0.46093412],
       [0.48017854],
       [0.39591849],
       [0.36551716],
       [0.41271239],
       [0.41910955],
       [0.42843985],
       [0.47157425],
       [0.45919627],
       [0.36261025],
       [0.46735523],
       [0.47135586],
       [0.37695291],
       [0.44492863],
       [0.45644756],
       [0.38494166],
       [0.48868326],
       [0.38161705],
       [0.42167866],
       [0.38994027],
       [0.3740443 ],
       [0.4844788 ],
       [0.49046083],
       [0.41480437],
       [0.40210915],
       [0.39809786],
```

(continues on next page)

(continued from previous page)

```
       [0.49195945],
       [0.41360141],
       [0.38396495],
       [0.43585678],
       [0.45175291],
       [0.38815235],
       [0.39152474],
       [0.42824886],
       [0.4691064 ],
       [0.3652643 ],
       [0.45632098],
       [0.47742713],
       [0.45361564],
       [0.39898102],
       [0.4806936 ],
       [0.44715114],
       [0.49595064],
       [0.46961672],
       [0.37299147],
       [0.43365596],
       [0.42415907],
       [0.47722936],
       [0.48771984],
       [0.47313675],
       [0.38212934],
       [0.45173258],
       [0.36681499],
       [0.45096277],
       [0.44605939]])
```

Step 2 - Compute the Loss

For a regression problem, the loss is given by the Mean Squared Error (MSE), that is, the average of all squared errors, that is, the average of all squared differences between labels (y) and predictions ($b + wx$).

For a regression problem, the loss is given by the Mean Squared Error (MSE), that is, the average of all squared errors, that is, the average of all squared differences between labels (y) and predictions (b + wx).

$$
\begin{aligned}
MSE &= \frac{1}{n}\sum_{i=1}^{n} error_i^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}(b + wx_i - y_i)^2
\end{aligned}
\tag{5.1}
$$

(5.4)

In the code below, we are using all data points of the training set to compute the loss, so $n = N = 80$, meaning we are performing batch gradient descent.

```
# Step 2 - Computing the loss
# We are using ALL data points, so this is BATCH gradient
# descent.
```

(continues on next page)

(continued from previous page)

```
# How wrong is our model? This is the error!
error = (yhat - y_train)
# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()
print(loss)
```

```
2.7421577700550976
```

Batch, Mini-batch, and Stochastic Gradient Descent

- if we use all points in the training set (n = N) to compute the loss, we are performing a batch gradient descent
- if we were to use a single point (n = 1) each time, it would be a stochastic gradient descent
- anything else (n) in-between 1 and N characterizes a minibatch gradient descent

Step 3 - Compute the gradient

A gradient is a partial derivative — why partial? Because one computes it with respect to (w.r.t.) a single parameter. We have two parameters, b and w, so we must compute two partial derivatives.

A derivative tells you how much a given quantity changes when you slightly vary some other quantity. In our case, how much does our MSE loss change when we vary each one of our two parameters separately?

```
# Step 3 - Computes gradients for both "b" and "w" parameters
b_grad = 2 * error.mean()
w_grad = 2 * (x_train * error).mean()
print(b_grad, w_grad)
```

```
-3.044811379650508 -1.8337537171510832
```

Step 4 - Update the Parameters

In the final step, we use the gradients to update the parameters. Since we are trying to minimize our losses, we reverse the sign of the gradient for the update. There is still another (hyper-)parameter to consider: the learning rate, denoted by the Greek letter alpha, which is the multiplicative factor that we need to apply to the gradient for the parameter update.

```
# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1
print(b, w)
# Step 4 - Updates parameters using gradients and
# the learning rate
b = b - lr * b_grad
w = w - lr * w_grad
print(b, w)
```

```
[0.49671415] [-0.1382643]
[0.80119529] [0.04511107]
```

Step 5 - Rinse and Repeat!

Now we use the updated parameters to go back to Step 1 and restart the process.

Definition of Epoch

An epoch is complete whenever every point in the training set (N) has already been used in all steps: forward pass, computing loss, computing gradients, and updating parameters.

```
n_epochs = 1000

for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent.
    # How wrong is our model? This is the error!
    error = (yhat - y_train)
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()

    # Step 3 - Computes gradients for both "b" and "w" parameters
    b_grad = 2 * error.mean()
    w_grad = 2 * (x_train * error).mean()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    b = b - lr * b_grad
    w = w - lr * w_grad

print(b, w)
```

```
[1.02354093] [1.96896412]
```

5.1.3 Sanity Check using Scikit-Learn

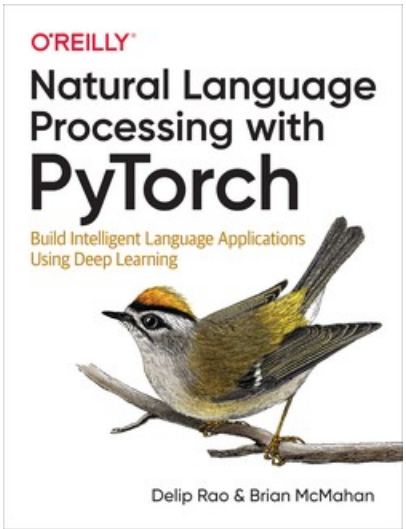
Just to make sure we haven't done any mistakes in our code, we can use Scikit- Learn's Linear Regression to fit the model and compare the coefficients.

```
from sklearn.linear_model import LinearRegression
# Sanity Check: do we get the same results as our
# gradient descent?
linr = LinearRegression()
linr.fit(x_train, y_train)
print(linr.intercept_, linr.coef_[0])
```

```
[1.02354075] [1.96896447]
```

5.2 Introduction to Pytorch Tensors

Pytorch is an optimised `tensor` manipulation library that offers an array of packages for deep learning. As compared to static frameworks such as Theano, Caffe and Tensorflow, Pytorch is in the family of dynamic frameworks, which does not require pre-defined computational graphs. This allows for a more flexible, imperative style of development, as it does not require the computational graphs to be first declared, compiled, and then excuted. However, this is potentially at the cost of computational efficiency, which makes it not as advantageous for production and mobile settings, but extremely useful during research and development.



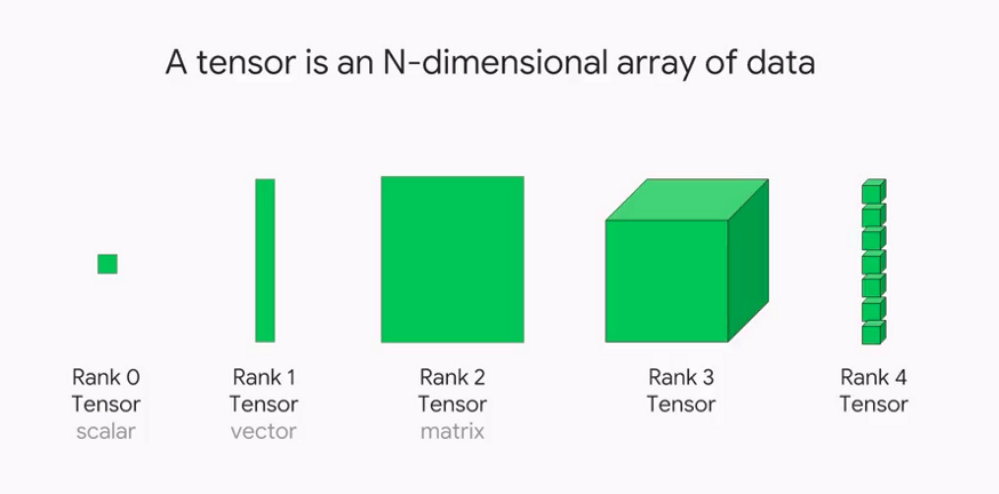
PyTorch

Reference: *Natural Lanuage Processing with PyTorch* - Building intelligent lanaguage applications using deep learning, by Delip Rao and Brian McMahan (copyright O'REILLY Feb 2019)

5.2.1 Tensors

Tensor

A tensor is a mathematical object holding some multidimensional data.



- A tensor of order zero is just a number, or a `scalar`.
- A tensor of order one (1st-order tensor) is an array of numbers, or a `vector`.
- A tensor of order two (2nd-order tensor) is an array of vectors, or a `matrix`.
- A tenosr of order n (nth-order tensor) is a generalised n-dimensional array of scalars.

5.2.2 Creating Tensors

You can create tensors in PyTorch pretty much the same way you create arrays in *Numpy*. Using `tensor()` you can create either a scalar or a tensor. PyTorch's tensors have equivalent functions as its Numpy counterparts, like: `ones()`, `zeros()`, `rand()`, `randn()` and many more. In the example below, we create one of each: scalar, vector, matrix and tensor or, saying it differently, one scalar and three tensors.

```
import torch
```

```
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
# two (2) 3x4 matrices
tensor = torch.randn((2, 3, 4), dtype=torch.float)
print(scalar)
print(vector)
print(matrix)
print(tensor)
```

```
tensor(3.1416)
tensor([1, 2, 3])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[[-0.1062,  0.0259,  1.0003,  1.7506],
         [ 1.5519,  0.5936,  0.0977,  1.0550],
         [-0.8436, -0.2855,  0.0348, -1.2236]],

        [[-0.7193, -1.7311,  0.3547, -2.5190],
         [ 0.6258, -0.6508, -0.4601, -0.4112],
         [ 0.6510,  0.2963, -0.1989,  0.3999]]])
```

A helper function `describe(x)`

Given a torch tensor `x`, we can use either `x.size()` function or `x.shape` property to look at the dimensionality of the torch tensor.

Note: `tensor.shape` is a property, not a callable function, whereas `tensor.size()` is a function.

```
def describe(x):
    print("Type:{}".format(x.type()))
    print("Shape:{}".format(x.shape))
    print("Size():{}".format(x.size()))
    print("Values: \n{}".format(x))
```

```
describe(scalar)
```

```
Type:torch.FloatTensor
Shape:torch.Size([])
Size():torch.Size([])
Values:
3.141590118408203
```

```
describe(vector)
```

```
Type:torch.LongTensor
Shape:torch.Size([3])
Size():torch.Size([3])
Values:
tensor([1, 2, 3])
```

```
describe(matrix)
```

```
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
describe(tensor)
```

```
Type:torch.FloatTensor
Shape:torch.Size([2, 3, 4])
Size():torch.Size([2, 3, 4])
Values:
tensor([[[-0.1062,  0.0259,  1.0003,  1.7506],
         [ 1.5519,  0.5936,  0.0977,  1.0550],
         [-0.8436, -0.2855,  0.0348, -1.2236]],

        [[-0.7193, -1.7311,  0.3547, -2.5190],
         [ 0.6258, -0.6508, -0.4601, -0.4112],
         [ 0.6510,  0.2963, -0.1989,  0.3999]]])
```

Creating a tensor with `torch.Tensor()`

```
describe(torch.Tensor(2,3))
```

```
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[0.0000e+00, 0.0000e+00, 2.1019e-44],
        [0.0000e+00, 1.4013e-45, 0.0000e+00]])
```

Warning: You might have noted that we can create tensors using both `torch.tensor()` and `torch.Tensor()`, note the subtle difference between the case of letter "t".

`torch.Tensor` is an alias for `torch.FloatTensor`, which creates tensors of float type.

`torch.tensor` on the other hand, infers the `dtype` automatically, and allows explicit specification of `dtype` during creation.

So let's stick to `torch.tensor` instead.

Creating a randomly initialized tensor

```
import torch

describe(torch.rand(2,3))    # uniform random
describe(torch.randn(2,3))   # normal random
```

```
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[0.2609, 0.1867, 0.2250],
        [0.7788, 0.2673, 0.5694]])
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[ 0.5022, -1.1496, -0.6783],
        [ 0.7880, -0.0197,  1.7654]])
```

Creating a filled tensor

```
import torch

describe(torch.zeros(2,3))

x = torch.ones(2,3)
describe(x)

x.fill_(5)
describe(x)
```

```
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[0., 0., 0.],
        [0., 0., 0.]])
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[1., 1., 1.],
        [1., 1., 1.]])
Type:torch.FloatTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[5., 5., 5.],
        [5., 5., 5.]])
```

Creating and initialising a tensor from lists

Observe the type difference between the `torch.tensor()` and `torch.Tensor()`.

```
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])
describe(x)
```

```
Type:torch.LongTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])
describe(x)
```

```
Type:torch.LongTensor
Shape:torch.Size([2, 3])
Size():torch.Size([2, 3])
Values:
tensor([[1, 2, 3],
        [4, 5, 6]])
```

Creating and initialising a tensor from Numpy

`from_numpy()` automatically inherits input array dtype. On the other hand, `torch.Tensor` is an alias for `torch.FloatTensor`.

Therefore, if you pass int32 array to torch.Tensor, output tensor is float tensor and they wouldn't share the storage. torch.from_numpy gives you torch.LongTensor as expected.

```
import torch
import numpy as np

a = np.arange(10)
```

```
describe(torch.from_numpy(a))
```

```
Type:torch.IntTensor
Shape:torch.Size([10])
Size():torch.Size([10])
Values:
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=torch.int32)
```

```
describe(torch.Tensor(a))
```

```
Type:torch.FloatTensor
Shape:torch.Size([10])
Size():torch.Size([10])
Values:
tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
describe(torch.tensor(a))
```

```
Type:torch.IntTensor
Shape:torch.Size([10])
Size():torch.Size([10])
Values:
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=torch.int32)
```

Subtleties in Torch Memory Model

Another way of creating tensor is to use `torch.as_tensor()`. What's the difference between `torch.as_tensor()` and `torch.tensor()`?

<literal>torch.tensor()</literal>

`torch.tensor` always copies the data. For example, `torch.tensor(x)` is equivalent to `x.clone().detach()`.

<literal>torch.as_tensor()</literal>

`torch.as_tensor` always tries to avoid copies of the data. One of the cases where `as_tensor` avoids copying the data is if the original data is a numpy array.

It is not always necessary or a good idea to copy and create a new tensor, especially when the tensor is large.

Reshaping a tensor

The same subtle difference exists between the methods that change the shape of a tensor, i.e. `view()` and `reshape()`.

Important: The `view()` method only returns a tensor with the desired shape that shares the underlying data with the original tensor - it **DOES NOT** create a new, independent, tensor!

The `reshape()` method **may or may not** create a copy! The reasons behind this apparently weird behavior are beyond the scope of this section - but this behavior is the reason why view() is preferred :-)

Why does it matter? Using `view()`, we get the same tensor with a different shape, any modification to the reshaped tensor will change the original tensor.

```
# We get a tensor with a different shape but it still is
# the SAME tensor
same_matrix = matrix.view(1, 6)
# If we change one of its elements...
same_matrix[0, 1] = 2.
# It changes both variables: matrix and same_matrix
print(matrix)
print(same_matrix)
```

```
tensor([[1., 2., 1.],
        [1., 1., 1.]])
tensor([[1., 2., 1., 1., 1., 1.]])
```

To copy all data into a separate independent tensor in the memory, so future operations will not affect the orignal, we will need to use `new_tensor()` or `clone()` methods.

```
# We can use "new_tensor" method to REALLY copy it into a new one
different_matrix = matrix.new_tensor(matrix.view(1, 6))
# Now, if we change one of its elements...
different_matrix[0, 1] = 3.
# The original tensor (matrix) is left untouched!
# But we get a "warning" from PyTorch telling us
# to use "clone()" instead!
print(matrix)
print(different_matrix)
```

```
tensor([[1., 2., 1.],
        [1., 1., 1.]])
tensor([[1., 3., 1., 1., 1., 1.]])
```

```
C:\Users\wei\AppData\Local\Temp/ipykernel_10816/887542709.py:2: UserWarning: To copy␣
 ↪construct from a tensor, it is recommended to use sourceTensor.clone().detach() or␣
 ↪sourceTensor.clone().detach().requires_grad_(True), rather than tensor.new_
 ↪tensor(sourceTensor).
  different_matrix = matrix.new_tensor(matrix.view(1, 6))
```

Warning: As from the warning message, we in fact should use `matrix.clone().detach()` to copy a tensor into a new duplicate, rather than `matrix.new_tensor()`.

In summary, the preferred functions are:

- from a list of values, use `torch.tensor(values, dtype="")`
- from numpy, use `torch.from_numpy()` or `torch.as_tensor()` to avoid data copying.
- from numpy, use `torch.tensor()` to copy it into a new tensor.
- from an existing tensor, `sourceTensor.view()` to avoid copying.
- from an existing tensor, `sourceTensor.clone().detach()` for a fresh duplicate.

5.2.3 Tensor Slicing, Indexing and Joining

```
import torch
from functions import describe

x = torch.arange(6).view(2,3)
describe(x)
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2, 3])
Values:
tensor([[0, 1, 2],
        [3, 4, 5]])
```

Contiguous Indexing using `[:a, :b]`

The code below accesses up to row 1 but not including row 1, and up to col 2, but no including col 2.

```
describe(x[:1, :2])
```

```
Type:torch.LongTensor
Shape/size:torch.Size([1, 2])
Values:
tensor([[0, 1]])
```

Noncontiguous Indexing

Using function `torch.index_select()`, the code below accesses column (`dim=1`) indexed by 0 and 2.

```
indices = torch.LongTensor([0, 2])
describe(torch.index_select(x, dim=1, index=indices))
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2, 2])
Values:
tensor([[0, 2],
        [3, 5]])
```

You can duplicate the same row or column multiple times, by specifying the same index multiple times.

```
indices = torch.LongTensor([0, 0, 0])
describe(torch.index_select(x, dim=0, index=indices))
```

```
Type:torch.LongTensor
Shape/size:torch.Size([3, 3])
Values:
tensor([[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]])
```

Use indices directly `[inices_list, indices_list]` can also achieve the same outcome.

```
row_indices = torch.arange(2).long()
col_indices = torch.LongTensor([0,2])
describe(x[row_indices, col_indices])
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2])
Values:
tensor([0, 5])
```

```
describe(x[[0,1], [0,2]])
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2])
Values:
tensor([0, 5])
```

Concatenating Tensors

```
x = torch.arange(6).view(2,3)
describe(x)
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2, 3])
Values:
tensor([[0, 1, 2],
        [3, 4, 5]])
```

```
describe(torch.cat([x, x], dim=0))
```

```
Type:torch.LongTensor
Shape/size:torch.Size([4, 3])
Values:
tensor([[0, 1, 2],
        [3, 4, 5],
        [0, 1, 2],
        [3, 4, 5]])
```

```
describe(torch.cat([x, x], dim=1))
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2, 6])
Values:
tensor([[0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5]])
```

```
describe(torch.stack([x, x], dim=1))
```

```
Type:torch.LongTensor
Shape/size:torch.Size([2, 2, 3])
Values:
tensor([[[0, 1, 2],
         [0, 1, 2]],

        [[3, 4, 5],
         [3, 4, 5]]])
```

Linear Algebra on tensors: multiplication

```
x1 = torch.arange(6).view(2,3).float()
describe(x1)
```

```
Type:torch.FloatTensor
Shape/size:torch.Size([2, 3])
Values:
tensor([[0., 1., 2.],
        [3., 4., 5.]])
```

Warning: `torch.arange()` creates LongTensor, for `torch.mm()`, we need to convert the LongTensor to FloatTensor by using `x.float()`.

```
x2 = torch.ones(3,2)
x2[:, 1] += 1
describe(x2)
```

```
Type:torch.FloatTensor
Shape/size:torch.Size([3, 2])
Values:
tensor([[1., 2.],
        [1., 2.],
        [1., 2.]])
```

```
describe(torch.mm(x1, x2))
```

```
Type:torch.FloatTensor
Shape/size:torch.Size([2, 2])
Values:
tensor([[ 3.,  6.],
        [12., 24.]])
```

5.2.4 CUDA tensors

So far, we have only created CPU tensors. What does it mean? It means the data in the tensor is stored in the computer's main memory and any operations performed on it are going to be handled by its CPU (the Central Processing Unit, for instance, an Intel® Core™ i7 Processor). So, although the data is, technically speaking, in the memory, we're still calling this kind of tensor a CPU tensor.

A GPU (which stands for Graphics Processing Unit) is the processor of a graphics card. These tensors store their data in the graphics card's memory and operations on top of them are performed by the GPU.

If you have a graphics card from NVIDIA, you can use the power of its GPU to speed up model training. PyTorch supports the use of these GPUs for model training using CUDA (Compute Unified Device Architecture), which needs to be previously installed and configured (please refer to the Setup Guide for more information on this).

If you do have a GPU (and you managed to install CUDA), we're getting to the part where you get to use it with PyTorch. But, even if you do not have a GPU, you should stick around in this section anyway… why? First, you can use a free GPU from Google Colab and, second, you should always make your code GPU-ready, that is, it should automatically run in a GPU, if one is available.

```
print(torch.cuda.is_available())
```

```
True
```

```
# prefered method: device agnostic tensor instantiation

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

So, if you don't have a GPU, your device is called cpu. If you do have a **GPU**, your device is called **cuda** or **cuda:0**.

If you have multiple GPUs, and want to check how many GPUs it has, or which model they are, you can figure it out using `cuda.device_count()` and `cuda.get_device_name()`:

```
n_cudas = torch.cuda.device_count()
for i in range(n_cudas):
    print(torch.cuda.get_device_name(i))
```

```
NVIDIA GeForce RTX 3080
```

We can use `.to(device)` to turn our tensor to a GPU tensor if you have a GPU device.

```
x = torch.rand(3,2).to(device)
describe(x)
```

```
Type:torch.cuda.FloatTensor
Shape/size:torch.Size([3, 2])
Values:
tensor([[0.3817, 0.3665],
        [0.9877, 0.7927],
        [0.9034, 0.5782]], device='cuda:0')
```

Warning: Mixing CUDA tensors with CPU-bound tensors will lead to errors. This is because we need to ensure the tensors are on the same device.

```
y = torch.rand(3,2)
x + y
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_10816/1126695582.py in <module>
      1 y = torch.rand(3,2)
----> 2 x + y

RuntimeError: Expected all tensors to be on the same device, but found at least two␣
 ↪devices, cuda:0 and cpu!
```

```
cpu_device = torch.device("cpu")
x = x.to(cpu_device)
y = y.to(cpu_device)
x + y
```

```
tensor([[0.6276, 0.9583],
        [0.7592, 1.2605],
        [1.0946, 0.9480]])
```

Note: It is expensive to move data back and forth from the GPU. Best practice is to carry out as much computation on GPU as possible and then just transfering the final results to CPU.

5.3 Linear Models in Pytorch

Now that we have some basic knowledge of Torch tensors, let's see how we can implement the linear model earlier using Pytorch.

5.3.1 Data Preparation

Let's first repeat the same data preparation process, first generating a synthetic dataset of 100 data points, then perform an 80%:20% split as training and validation dataset, respectively.

```
import numpy as np
import torch
```

```
true_b = 1
true_w = 2
N = 100
# Data Generation
np.random.seed(42)
x = np.random.rand(N, 1)
# Guassian noise to add some randomness to y
epsilon = (.1 * np.random.randn(N, 1))
y = true_b + true_w * x + epsilon

# Shuffles the indices
idx = np.arange(N)
```

(continues on next page)

(continued from previous page)

```
np.random.shuffle(idx)
# Uses first 80 random indices for train
train_idx = idx[:int(N*.8)]
# Uses the remaining indices for validation
val_idx = idx[int(N*.8):]
# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

No matter you have a GPU or not, the best practice is to use `.to(device)` method to make your code GPU ready.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

```
# Here we can see the difference - notice that .type() is more
# useful since it also tells us WHERE the tensor is (device)
print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

```
<class 'numpy.ndarray'> <class 'torch.Tensor'> torch.cuda.FloatTensor
```

We normally can turn a tensor back to a *Numpy* array using `sourceTensor.numpy()`, but now we have GPU tensor, which cannot be directly handled by Numpy. We have turn it back to a CPU tensor first before converting to a Numpy array.

```
back_to_numpy = x_train_tensor.cpu().numpy()
```

Good Practice

It is a good practice to always first `cpu()` and then `numpy()`, even if you are using a CPU. It follows the same principle of to(device): you may share your code with others who may be using a GPU.

5.3.2 Creating Parameters

What distinguishes a tensor used for training data (or validation, or test) — like the ones we've just created — from a tensor used as a (trainable) parameter/weight?

The latter (a parameter) requires the computation of its gradients, so we can update their values (the parameters' values).

In Pytorch, we use the `requires_grad=True` argument to tell PyTorch to compute gradients for us.

A tensor for a learnable parameter requires a gradient!

Good Practice

To make GPU ready code, we should specify the device at the moment of creation to avoid shadowing the gradient requirement.

```
# We can specify the device at the moment of creation
# RECOMMENDED!
# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
print(b, w)
```

```
tensor([0.1940], device='cuda:0', requires_grad=True) tensor([0.1391], device='cuda:0
↪', requires_grad=True)
```

Note: Notice that even with the same seed value, because *Pytorch* and *Numpy* are two different packages, they have different implemenations of the `randn()` method, thus different results.

5.3.3 Autograd

In Pytorch, we don't need to worry about partial derivatives, chain rule, or anything like it. Autograd is PyTorch's automatic differentiation package.

`backward()`

To tell PyTorch to compute all gradients, we use the `backward()` method. It will compute gradients for all (requiring gradient) tensors involved in the computation of a given variable.

Recall that we need to compute the partial derivatives of the loss function w.r.t. our parameters. Hence, we need to invoke the `backward()` method from the corresponding Python variable: `loss.backward()`.

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train_tensor
# Step 2 - Computes the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train_tensor)
# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()
# Step 3 - Computes gradients for both "b" and "w" parameters
# No more manual computation of gradients!
# b_grad = 2 * error.mean()
# w_grad = 2 * (x_tensor * error).mean()
loss.backward()
```

We have set `requires_grad=True` to both `b` and `w`, so they are obviously included in the list of gradient calculation. We use them both to compute `yhat`, so it will also make it to the list. Then we use `yhat` to compute the `error`, so `error` is also on the list.

`x_train_tensor` and `y_train_tensor` however, are not gradient-requiring tensors, so `backward()` does not care about them.

```
print(error.requires_grad, yhat.requires_grad, b.requires_grad, w.requires_grad)
print(y_train_tensor.requires_grad, x_train_tensor.requires_grad)
```

```
True True True True
False False
```

grad

We can inspect the actual values of the gradients by looking at the `grad` attribute of a tensor.

```
print(b.grad, w.grad)
```

```
tensor([-3.3881], device='cuda:0') tensor([-1.9439], device='cuda:0')
```

Accumulated Gradients

Let's run the backward function again:

```
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train_tensor
# Step 2 - Computes the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train_tensor)
# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()
# Step 3 - Computes gradients for both "b" and "w" parameters
# No more manual computation of gradients!
# b_grad = 2 * error.mean()
# w_grad = 2 * (x_tensor * error).mean()
loss.backward()
```

```
print(b.grad, w.grad)
```

```
tensor([-6.7762], device='cuda:0') tensor([-3.8878], device='cuda:0')
```

Note: If we ran this above code again, the gradient of b and w exactly doubled. This is because Pytorch implements an accumlated gradients to circumvent hardware limitations. If a minibatch is still too big to fit in memory, we can split it further into "subminibatch", that's when the aggregated gradients become useful.

zero_

For training problem that does not have memory limitations, every time we use the gradients to update the parameters, we need to zero the gradients afterward. This what `zero_()` is good for.

```
# This code will be placed _after_ Step 4
# (updating the parameters)
b.grad.zero_(), w.grad.zero_()
```

```
(tensor([0.], device='cuda:0'), tensor([0.], device='cuda:0'))
```

Important: In PyTorch, every method that ends with an underscore (_), like the `requires_grad_()` and `zero_()` method above, makes changes in-place, in other words, they will modify the underlying variable.

5.3.4 Put it all together

```
# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train_tensor

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent. How wrong is our model? That's the error!
    error = (yhat - y_train_tensor)
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()

    # Step 3 - Computes gradients for both "b" and "w"
    # parameters. No more manual computation of gradients!
    # b_grad = 2 * error.mean()
    # w_grad = 2 * (x_tensor * error).mean()
    # We just tell PyTorch to work its way BACKWARDS
    # from the specified loss!
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate. But not so fast...
    # FIRST ATTEMPT - just using the same code as before
    # AttributeError: 'NoneType' object has no attribute 'zero_'
    # b = b - lr * b.grad
    # w = w - lr * w.grad
    # print(b)

    # SECOND ATTEMPT - using in-place Python assingment
    # RuntimeError: a leaf Variable that requires grad
    # has been used in an in-place operation.
    # b -= lr * b.grad
    # w -= lr * w.grad

    # THIRD ATTEMPT - NO_GRAD for the win!
    # We need to use NO_GRAD to keep the update out of
    # the gradient computation. Why is that? It boils
    # down to the DYNAMIC GRAPH that PyTorch uses...
    with torch.no_grad():
```

(continues on next page)

(continued from previous page)

```
        b -= lr * b.grad
        w -= lr * w.grad

    # PyTorch is "clingy" to its computed gradients, we
    # need to tell it to let it go...
    b.grad.zero_()
    w.grad.zero_()

print(b, w)
```

```
tensor([1.0235], device='cuda:0', requires_grad=True) tensor([1.9690], device='cuda:0
↪', requires_grad=True)
```

In the first attempt, if we use the same update structure as in our Numpy code, we'll get a weird error but we can get a hint of what's going on by looking at the tensor itself — once again, we "lost" the gradient while reassigning the update results to our parameters. Thus, the grad attribute turns out to be None, and it raises the error…

Important: We use `with torch.no_grad():` to ensure the update is not tracked by the *dyanmic computation graph* mechanism of Pytorch. We will talk about compuation graph next lab.

CHAPTER

SIX

LAB06: NEURAL NETWORK BUILDING BLOCKS

Each node in a neural network is called a **perceptron** unit, which has three "knobs", a set of weights (w), a bias (b), and an activation function (f). The weights and bias are learned from the data, and the activation function is hand picked depending on the network designer's intuition of the network and its target outputs. Mathematically,

$y = f(wx + b)$

Listing 6.1: A skeleton of a Perceptron

```
class Perceptron(nn.Module):
    """
    A perceptron is one linear layer
    """

    def __init__(self, input_dim):
        """
        Args:
            input_dim (int): size of the input features
        """
        super(Perceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, 1)

    def forward(self, x_in):
        """The forward pass of the perceptron

        Args:
            x_in (torch.Tensor): an input data tensor
                x_in.shape should be (batch, num_features)
        Returns:
            the resulting tensor. tensor.shape should be (batch,).
        """
        return torch.sigmoid(self.fc1(x_in))
```

In this lab, we will look at the activation functions, and loss functions in more detail, and finish with building and train a simple neural network in PyTorch for simulating the XOR binary operator with Object-orientation in mind.

Reference: *Deep Learning in PyTorch, Step by Step A Beginner's Guide* by Daniel Voigt Godoy - Chapter 2 and 3.

6.1 Activation Functions and their derivatives

Activation functions are salient to provide the important non-linearities to Neural Networks, which turn a linear model into powerful scalable models that are fundamental to modern neural computation.

This notebook visualises the popular activation functions and their derivatives, adapted from this reference blog on activation functions

6.1.1 Sigmoid Function

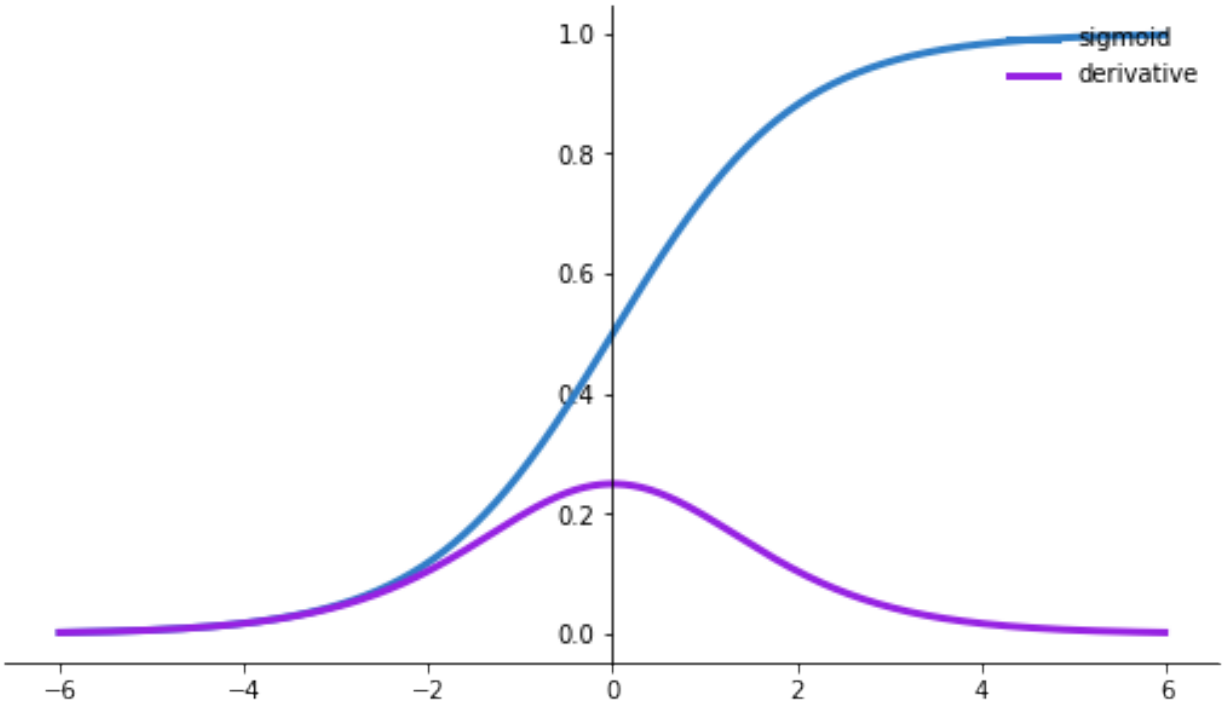
$$t = f(z) = \frac{1}{1+e^{-z}}$$

$$\frac{dt}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = t * (1-t)$$

In Plain Python with Numpy

```
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(x):
    s=1/(1+np.exp(-x))
    ds=s*(1-s)
    return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3, label="sigmoid")
ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

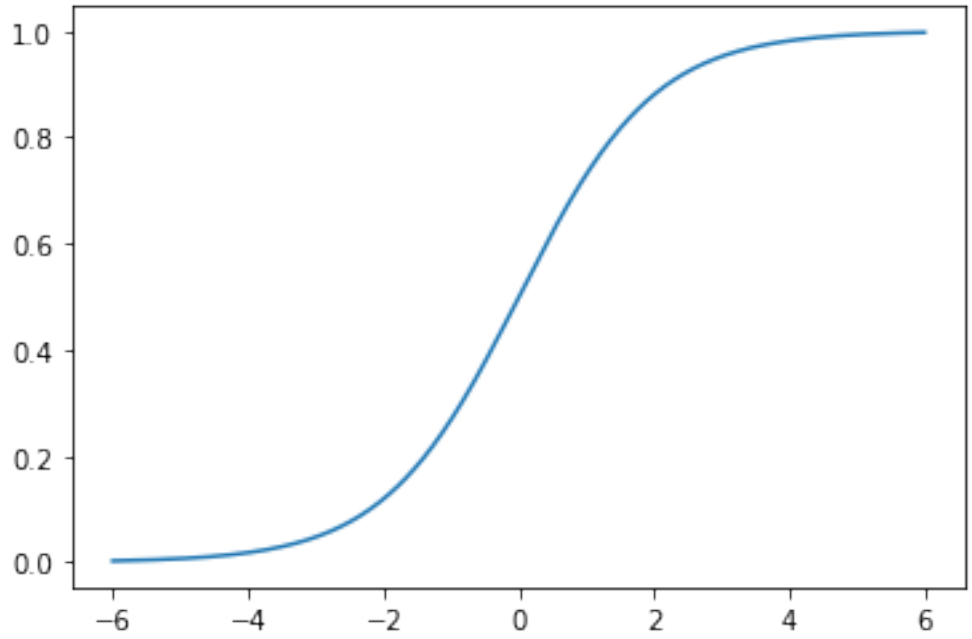
```
C:\Users\wei\AppData\Local\Temp/ipykernel_15812/3857614770.py:21: UserWarning:␣
 ↪Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a␣
 ↪non-GUI backend, so cannot show the figure.
  fig.show()
```



In Pytorch with `torch.sigmoid()`

Warning: `torch.range` is deprecated and will be removed in a future release because its behavior is inconsistent with Python's range builtin. Instead, use torch.arange, which produces values in [start, end). Do not use: `x = torch.range(-6,6,0.01)`

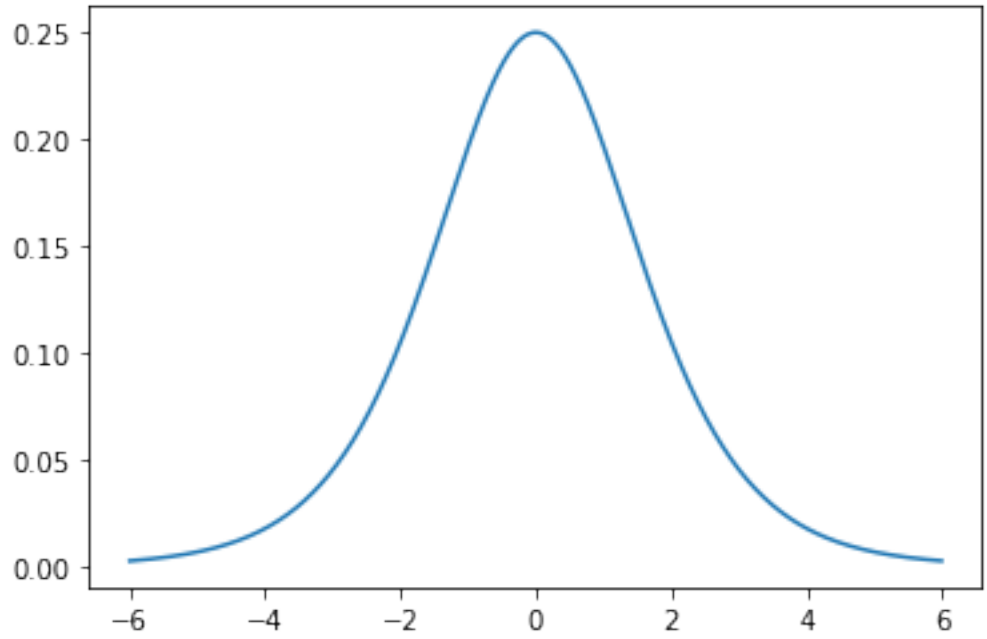
```
import torch
import numpy
import matplotlib.pyplot as plt

x = torch.arange(-6,6,0.01)
y = torch.sigmoid(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```



[AUTOGRAD] (https://pytorch.org/tutorials/beginner/former_torchies/autograd_tutorial.html) is a core Torch package for automatic differentiation.

```
x = torch.arange(-6,6,0.01)
x.requires_grad_()
print(x[1:10])
t = torch.sigmoid(x)

t.backward(torch.ones(x.shape))

# x.grad the gradient at each x with respect to function t
dt = x.grad
plt.plot(x.detach().numpy(), dt.detach().numpy())
plt.show()
```

```
tensor([-5.9900, -5.9800, -5.9700, -5.9600, -5.9500, -5.9400, -5.9300, -5.9200,
        -5.9100], grad_fn=<SliceBackward>)
```




```
x
```

```
tensor([-6.0000, -5.9900, -5.9800,  ...,  5.9700,  5.9800,  5.9900],
       requires_grad=True)
```

6.1.2 Hyperbolic tanh

$$t = tanh(z) = \frac{e^z - e^{-z}}{e^z - e^{-z}}$$

$$\frac{dt}{dz} = 1 - t^2$$

In Plain Python and Numpy

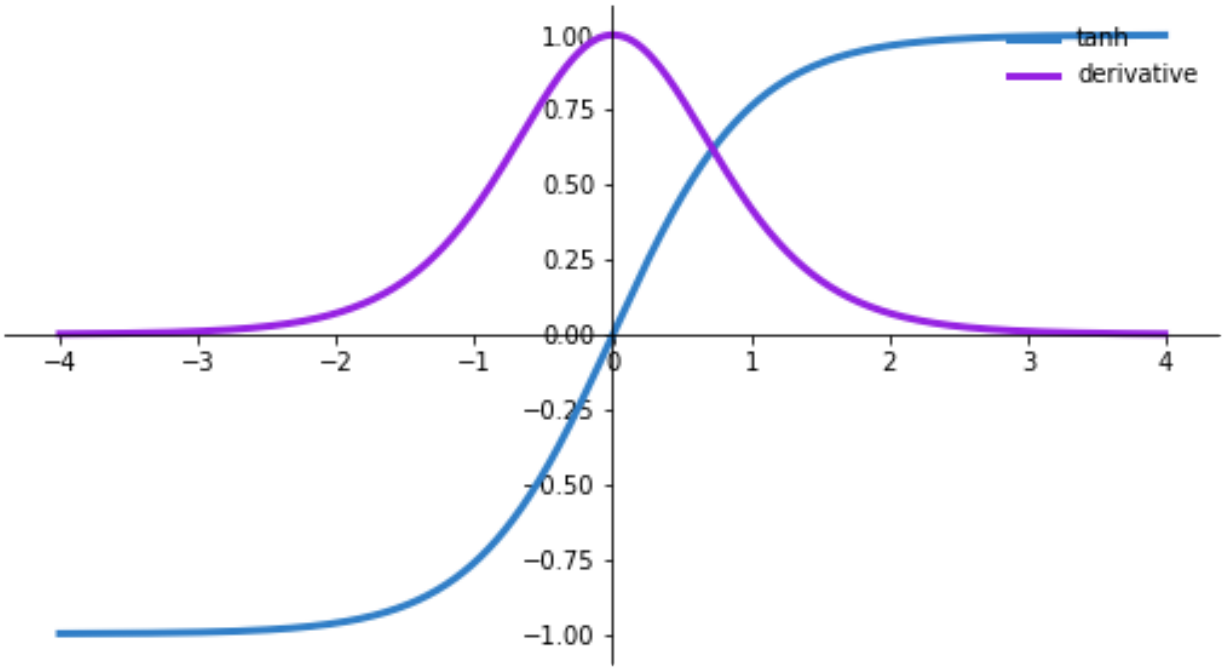
```
import matplotlib.pyplot as plt
import numpy as np

def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    dt=1-t**2
    return t,dt


z=np.arange(-4,4,0.01)
# print(tanh(z)[0])
tanh(z)[0].size,tanh(z)[1].size
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
```

(continues on next page)

(continued from previous page)

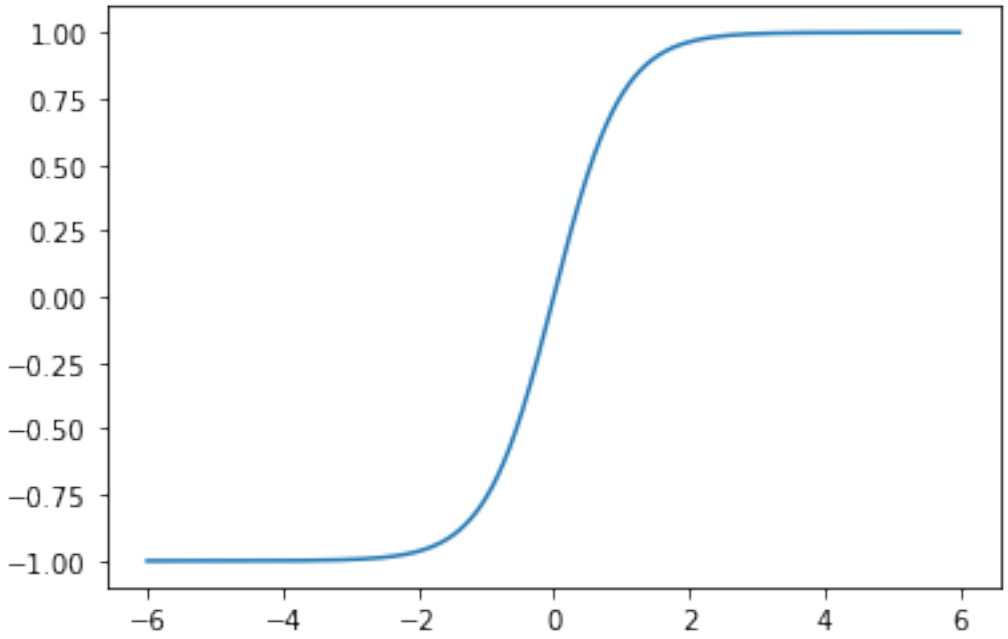
```
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")
ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

```
C:\Users\wei\AppData\Local\Temp/ipykernel_15812/1006845169.py:25: UserWarning:␣
 ↪Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a␣
 ↪non-GUI backend, so cannot show the figure.
  fig.show()
```

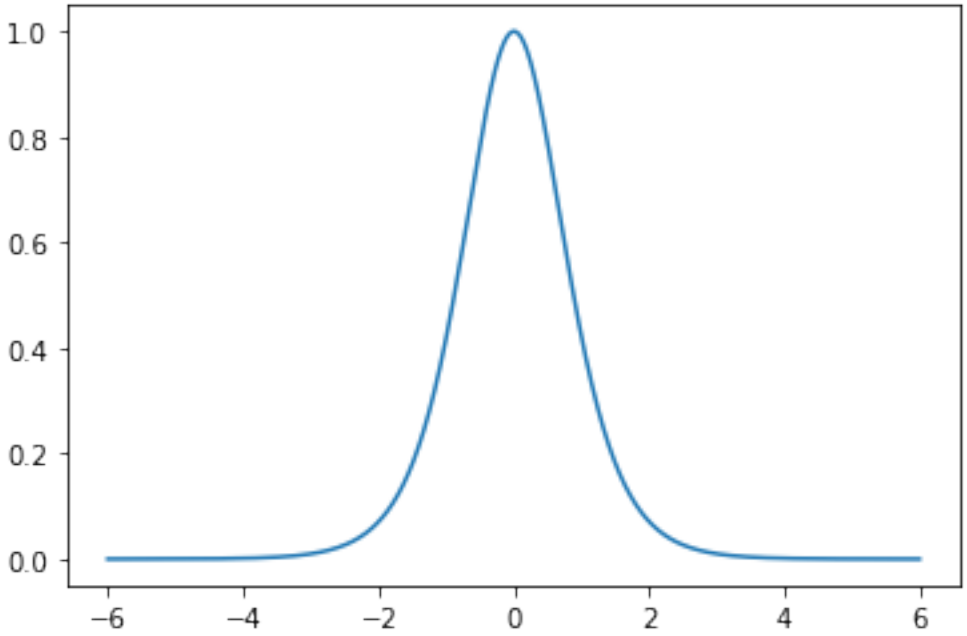


In Pytorch with `torch.tanh()`

```
import torch
import matplotlib.pyplot as pyplot

x = torch.arange(-6, 6, 0.01)
y = torch.tanh(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```




```
x = torch.arange(-6,6,0.01)
x.requires_grad_()
y = torch.tanh(x)

y.backward(torch.ones(x.shape))

plt.plot(x.detach().numpy(), x.grad.detach().numpy())
plt.show()
```



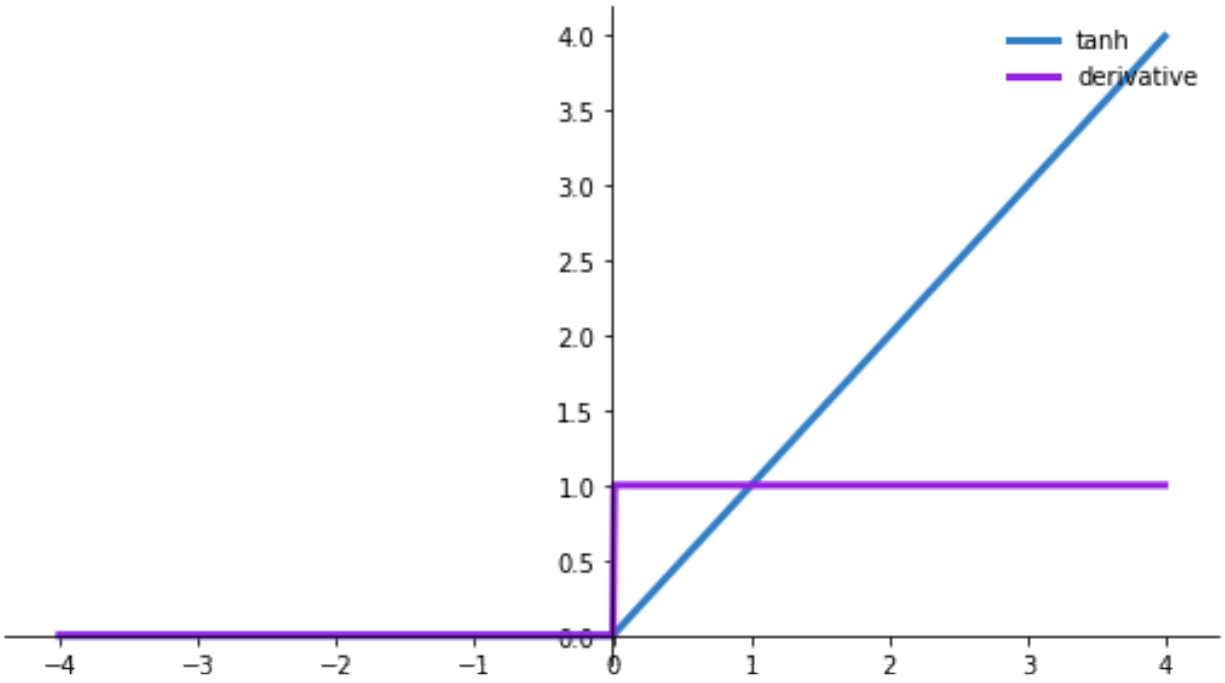
6.1.3 Relu

$$t = f(z) = max(0, z)$$

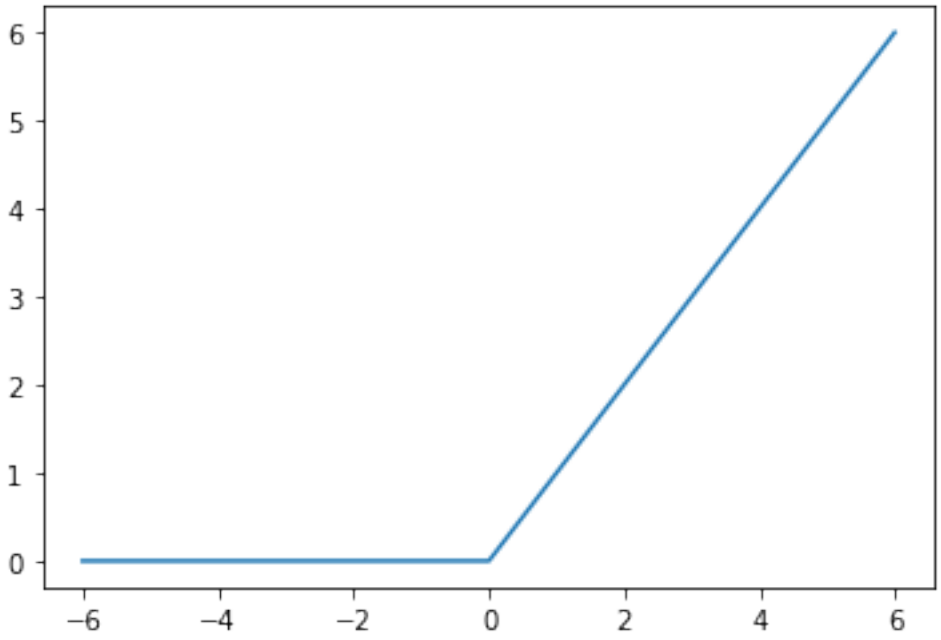
$$\frac{dt}{dz} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

```
import matplotlib.pyplot as plt
import numpy as np

def relu(x):
    t = [v if v >= 0 else 0 for v in x]
    dt = [1 if v >= 0 else 0 for v in x]
    t = np.array(t)
    dt = np.array(dt)
    return t,dt

z=np.arange(-4,4,0.01)
#print(relu(z)[0])
relu(z)[0].size,relu(z)[1].size
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(z,relu(z)[0], color="#307EC7", linewidth=3, label="tanh")
ax.plot(z,relu(z)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

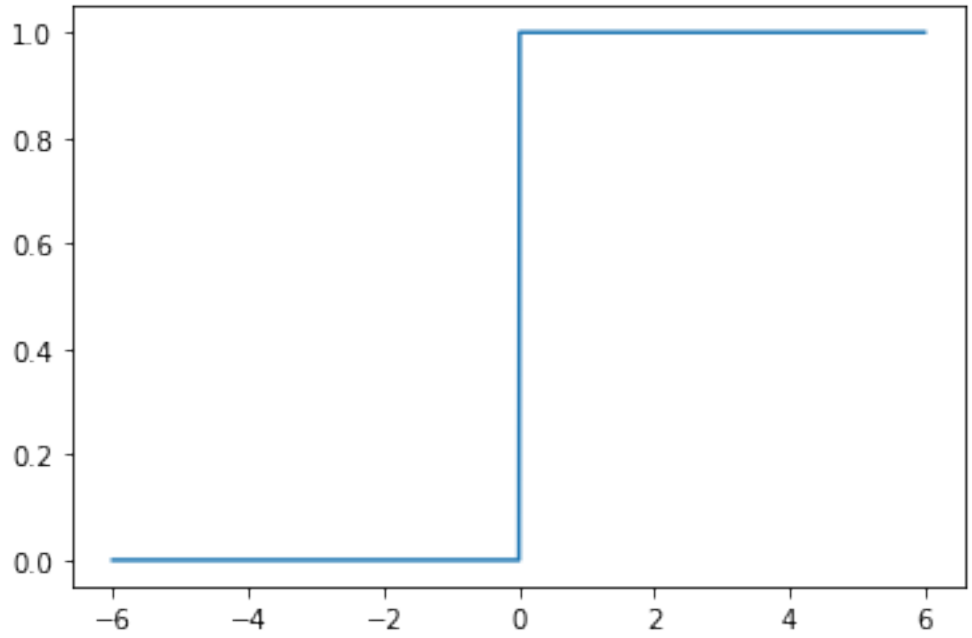
```
C:\Users\wei\AppData\Local\Temp/ipykernel_15812/1318022484.py:26: UserWarning:␣
 ↪Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a␣
 ↪non-GUI backend, so cannot show the figure.
  fig.show()
```




```
import torch
import matplotlib.pyplot as plt

relu = torch.nn.ReLU()
x = torch.arange(-6, 6, 0.01)
y = relu(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```



```
x = torch.arange(-6,6,0.01)
x.requires_grad_()
# y = torch.nn.ReLU not working
y = torch.nn.functional.relu(x)

y.backward(torch.ones(x.shape))

plt.plot(x.detach().numpy(), x.grad.detach().numpy())
plt.show()
```



6.1.4 Parametric and Leaky Relu

Dying ReLU

The clipping effect of ReLU that helps with the vanishing gradient problem can also become an issue. Over time, certain ouputs in the network can simply vecome zero and never revive again. The is called the *dying ReLU* problem.

The Prametric ReLU (PReLU) is introduced to address the dying ReLU, where the leak coefficient a is a learned parameter.

$$t = max(z, a * z) = f(a, z) \begin{cases} z & \text{if } z \geq 0 \\ a * z & \text{otherwise} \end{cases}$$

$$\frac{dt}{dz} = \begin{cases} 1 & \text{if } z \geq 0 \\ a & \text{otherwise} \end{cases}$$

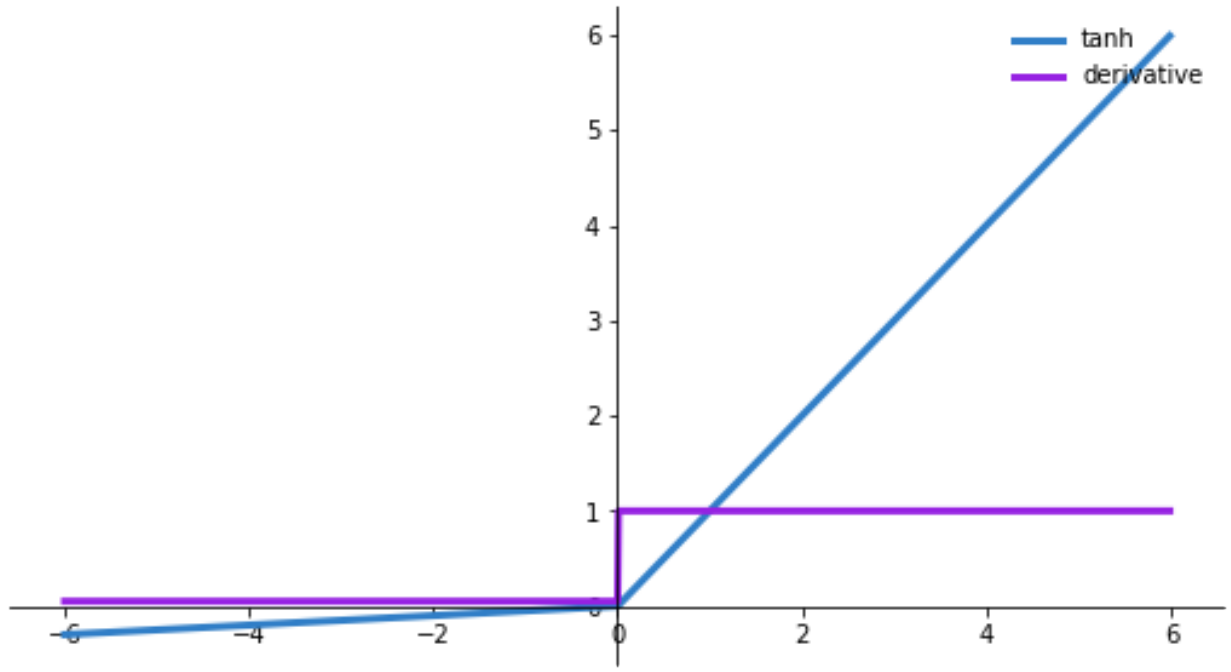
```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
def parametric_relu(a, x):
    t = [v if v >= 0 else a*v for v in x]
    dt = [1 if v >= 0 else a for v in x]
    t = np.array(t)
    dt = np.array(dt)
    return t,dt

z=np.arange(-6,6,0.01)
#print(relu(z)[0])
t=parametric_relu(0.05,z)
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(z,t[0], color="#307EC7", linewidth=3, label="tanh")
ax.plot(z,t[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

```
C:\Users\wei\AppData\Local\Temp/ipykernel_15812/586447431.py:26: UserWarning:␣
 ↪Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a␣
 ↪non-GUI backend, so cannot show the figure.
  fig.show()
```

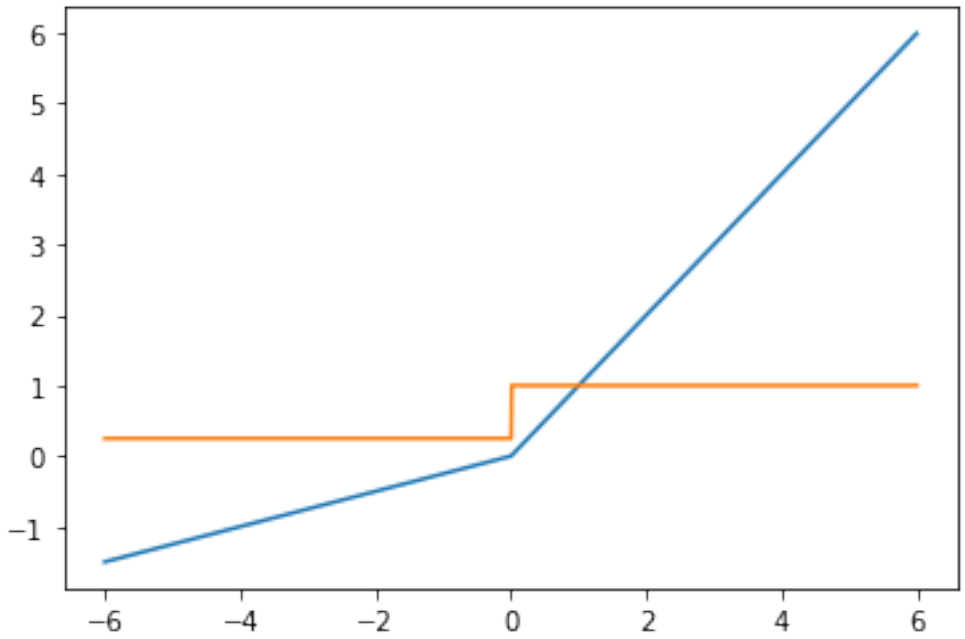



```
import torch
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
prelu = torch.nn.PReLU(num_parameters=1)
x = torch.arange(-6, 6, 0.01)
x.requires_grad_()
y = prelu(x)
y.backward(torch.ones(x.shape))
plt.plot(x.detach().numpy(), y.detach().numpy())
plt.plot(x.detach().numpy(), x.grad.detach().numpy())
plt.show()
```



6.1.5 Softmax

The softmax function squahses the output of each unit to be between 0 and 1, like sigmod function. However, the softmax operation also divides each output by the sum of all outputs, which gives a discrete probabilty distribution over k possible classes.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

```
import torch.nn as nn
import torch

softmax = nn.Softmax(dim=1)
x_input = torch.randn(1, 3)
y_output = softmax(x_input)
print(x_input)
print(y_output)
print(torch.sum(y_output, dim=1))
```

```
tensor([[-0.0027,  1.0101, -0.9370]])
tensor([[0.2412, 0.6641, 0.0947]])
tensor([1.])
```

For more, see this blog on Activation Functions in Neural Networks.

6.2 Loss Functions

```
import numpy as np
import torch
import torch.nn as nn

seed = 42

torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
np.random.seed(seed)

%matplotlib inline
```

6.2.1 Perceptron

Each node in a neural network is called a perceptron unit, which has three "knobs", a set of weights (w), a bias (b), and an activation function (f). The weights and bias are learned from the data, and the activation function is hand picked depending on the network designer's intuition of the network and its target outputs. Mathematically,

$y = f(wx + b)$

```
class Perceptron(nn.Module):
    """
    A perceptron is one linear layer
    """

    def __init__(self, input_dim):
        """
        Args:
            input_dim (int): size of the input features
        """
        super(Perceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, 1)

    def forward(self, x_in):
        """The forward pass of the perceptron

        Args:
            x_in (torch.Tensor): an input data tensor
                x_in.shape should be (batch, num_features)
        Returns:
            the resulting tensor. tensor.shape should be (batch,).
        """
        return torch.sigmoid(self.fc1(x_in)).squeeze()
```

What does .squeeze() do in Python?

```
# Python program explaining
# numpy.squeeze function

import numpy as np
```

(continues on next page)

(continued from previous page)

```
in_arr = np.array([[[2, 2, 2], [2, 2, 2]]])

print ("Input array : ", in_arr)
print("Shape of input array : ", in_arr.shape)

out_arr = np.squeeze(in_arr)

print ("output squeezed array : ", out_arr)
print("Shape of output array : ", out_arr.shape)
```

```
Input array :  [[[2 2 2]
  [2 2 2]]]
Shape of input array :  (1, 2, 3)
output squeezed array :  [[2 2 2]
 [2 2 2]]
Shape of output array :  (2, 3)
```

6.2.2 Softmax

```
softmax = nn.Softmax(dim=1)
x_input = torch.randn(1, 3)
y_output = softmax(x_input)
print(x_input)
print(y_output)
print(torch.sum(y_output, dim=1))
```

```
tensor([[0.3367, 0.1288, 0.2345]])
tensor([[0.3683, 0.2992, 0.3325]])
tensor([1.])
```

6.2.3 MSELoss

```
import torch
import torch.nn as nn

mse_loss = nn.MSELoss()
torch.manual_seed(42)
yhat = torch.randn(3, 5, requires_grad=True)
y = torch.randn(3, 5)
loss = mse_loss(yhat, y)
loss.backward()
print(loss)
```

```
tensor(1.0192, grad_fn=<MseLossBackward>)
```

6.2.4 MAE Loss - L1Loss

```
import torch
import torch.nn as nn

mse_loss = nn.L1Loss()
torch.manual_seed(42)
yhat = torch.randn(3, 5, requires_grad=True)
y = torch.randn(3, 5)
loss = mse_loss(yhat, y)
print(yhat)
print(y)
loss.backward()
print(loss)
```

```
tensor([[ 0.3367,  0.1288,  0.2345,  0.2303, -1.1229],
        [-0.1863,  2.2082, -0.6380,  0.4617,  0.2674],
        [ 0.5349,  0.8094,  1.1103, -1.6898, -0.9890]], requires_grad=True)
tensor([[ 0.9580,  1.3221,  0.8172, -0.7658, -0.7506],
        [ 1.3525,  0.6863, -0.3278,  0.7950,  0.2815],
        [ 0.0562,  0.5227, -0.2384, -0.0499,  0.5263]])
tensor(0.8502, grad_fn=<L1LossBackward>)
```

6.2.5 RMSE Loss

```
class RMSELoss(nn.Module):
    def __init__(self):
        super().__init__()
        self.mse = nn.MSELoss()

    def forward(self,yhat,y):
        eps = 1e-7
        return torch.sqrt(self.mse(yhat,y) + eps)

criterion = RMSELoss()

torch.manual_seed(42)
yhat = torch.randn(3, 5, requires_grad=True)
y = torch.randn(3, 5)
loss = criterion(yhat,y)
print(loss)
```

```
tensor(1.0096, grad_fn=<SqrtBackward>)
```

6.2.6 CrossEntropyLoss

```
import torch
import torch.nn as nn

ce_loss = nn.CrossEntropyLoss()

torch.manual_seed(42)
outputs = torch.randn(3, 5, requires_grad=True)
targets = torch.tensor([1, 0, 3], dtype=torch.int64)
loss = ce_loss(outputs, targets)
loss.backward()
print (loss)
```

```
tensor(2.6812, grad_fn=<NllLossBackward>)
```

Cross Entropy Loss - Manual Calculation

```
labels = torch.tensor([1.0, 0.0])
predictions = torch.tensor([.9, .2])
# Positive class (labels == 1)
positive_pred = predictions[labels == 1]
first_summation = torch.log(positive_pred).sum()
# Negative class (labels == 0)
negative_pred = predictions[labels == 0]
second_summation = torch.log(1 - negative_pred).sum()
# n_total = n_pos + n_neg
n_total = labels.size(0)
loss = -(first_summation + second_summation) / n_total
loss
```

```
tensor(0.1643)
```

```
positive_pred = predictions[labels == 1]
print(labels == 1)
print(positive_pred)
```

```
tensor([ True, False])
tensor([0.9000])
```

```
summation = torch.sum(labels * torch.log(predictions) + (1 - labels) * torch.log(1 -␣
↪predictions))
loss = -summation / n_total
loss
```

```
tensor(0.1643)
```

Binary Cross Entropy Loss - PyTorch

```
bce_loss = nn.BCELoss()
sigmoid = nn.Sigmoid()

torch.manual_seed(42)
probabilities = sigmoid(torch.randn(4, 1, requires_grad=True))
print(probabilities)

targets = torch.tensor([1, 0, 1, 0], dtype=torch.float32).view(4, 1)
loss = bce_loss(probabilities, targets)
loss.backward()
print(loss)
```

```
tensor([[0.5834],
        [0.5322],
        [0.5583],
        [0.5573]], grad_fn=<SigmoidBackward>)
tensor(0.6741, grad_fn=<BinaryCrossEntropyBackward>)
```

```
labels = torch.tensor([1.0, 0.0])
predictions = torch.tensor([.9, .2])
# RIGHT
right_loss = bce_loss(predictions, labels)
# WRONG
wrong_loss = bce_loss(labels, predictions)
print(right_loss, wrong_loss)
```

```
tensor(0.1643) tensor(15.0000)
```

BCE with Logits Loss

```
import numpy as np
import torch.nn as nn

def log_odds_ratio(prob):
    return np.log(prob/(1-prob))
```

```
logit1 = log_odds_ratio(.9)
logit2 = log_odds_ratio(.2)
labels = torch.tensor([1.0, 0.0])
logits = torch.tensor([logit1, logit2])
print(logits)
loss_fn_logits = nn.BCEWithLogitsLoss()
loss = loss_fn_logits(logits, labels)
loss
```

```
tensor([ 2.1972, -1.3863], dtype=torch.float64)
```

```
tensor(0.1643)
```

6.2.7 Negative Log-Likelihood Loss

Log Softmax

```
import torch.nn as nn
import torch.nn.functional as F
```

```
logits = torch.tensor([ 1.3863, 0.0000, -0.6931])
```

Compute Softmax Manually

```
# e^z is odds ratios
odds_ratios = torch.exp(logits)
odds_ratios
```

```
tensor([4.0000, 1.0000, 0.5000])
```

```
softmaxed = odds_ratios/odds_ratios.sum()
softmaxed
```

```
tensor([0.7273, 0.1818, 0.0909])
```

Compute Softmax in PyTorch

PyTorch provide the typical implementations both as a function (`F.softmax`) and as a module (`nn.Softmax`), which both take in logits (the log odds) as input. We need to tell the softmax function which dimension should be applied to.

In general, our models will produce logits with the shape `(number of data points, number of classes)`, so the right dimension to apply softmax to is the last one (dim=-1).

```
## Using module
softmaxed_torch = nn.Softmax(dim=-1)(logits)
softmaxed_torch
```

```
tensor([0.7273, 0.1818, 0.0909])
```

```
# Using functional
softmaxed_torch = F.softmax(logits, dim=-1)
softmaxed_torch
```

```
tensor([0.7273, 0.1818, 0.0909])
```

6.2.8 LogSoftmax

```
torch.log(softmaxed)
```

```
tensor([-0.3185, -1.7048, -2.3979])
```

```
# Use functional
log_probs = F.log_softmax(logits, dim=-1)
log_probs
```

```
tensor([-0.3185, -1.7048, -2.3979])
```

```
# Use module
log_probs = nn.LogSoftmax(dim=-1)(logits)
log_probs
```

```
tensor([-0.3185, -1.7048, -2.3979])
```

6.2.9 Negative Loglikelihood Loss

NLL loss is a summation of the negative log probilities (log softmax) for all classes over the data points.

Let's take a look at the following 3 classes, 5 data points' NLL loss calculation.

logits → log softmax (log probability) → NLL

```
# manual implementation

torch.manual_seed(11)
logits = torch.randn((5, 3))
labels = torch.tensor([0, 0, 1, 2, 1])
log_probs = F.log_softmax(logits, dim=-1)
log_probs
```

```
tensor([[-1.5229, -0.3146, -2.9600],
        [-1.7934, -1.0044, -0.7607],
        [-1.2513, -1.0136, -1.0471],
        [-2.6799, -0.2219, -2.0367],
        [-1.0728, -1.9098, -0.6737]])
```

NLL is a negated sum of log probabilities.

Manual Calculation

```
#indices = torch.tensor([[0], [0], [1], [2], [1]])
indices = torch.tensor([[idx] for idx in labels])
log_probs_cls = log_probs.gather(1, indices)
```

```
# for each data point, we select the log_probs of corresponding class
-log_probs_cls.mean()
```

```
tensor(1.6553)
```

Using PyTorch

```
F.nll_loss(log_probs, labels)
```

```
tensor(1.6553)
```

```
loss_fn = nn.NLLLoss()
loss_fn(log_probs, labels)
```

```
tensor(1.6553)
```

The preferred module implementation `nn.NLLLoss` loss function is a higher-order function, and this one takes three optional arguments (the others are deprecated and you can safely ignore them).

reduction: it takes either `mean`, `sum`, or `none`. The default, `mean`, corresponds to our equation above. As expected, `sum` will return the sum of the errors, instead of the average. The last option, `none`, corresponds to the unreduced form, that is, it returns the full array of errors.

weight: it takes a tensor of length C, that is, containing as many weights as there are classes.

Important: this weight argument can be used to handle imbalanced datasets, unlike the weight argument in the binary cross-entropy losses we've seen before. Also, unlike the pos_weight argument of BCEWithLogitsLoss, the NLLLoss computes a true weighted average when this argument is used.

---

ignore_index: it takes one integer, corresponding to the one (and only one) class index that should be ignored when computing the loss. It can be used to mask a particular label that is not relevant to the classification task.

Example of Using Weight

What if we want to balance our dataset, giving data points with label (y=2) double the weight of the other classes?

```
loss_fn = nn.NLLLoss(weight=torch.tensor([1., 1., 2.]))
loss_fn(log_probs, labels)
```

```
tensor(1.7188)
```

Example of Using ignore_index

What if we want to simply ignore data points with label (y=2)?

```
loss_fn = nn.NLLLoss(ignore_index=2)
loss_fn(log_probs, labels)
```

```
tensor(1.5599)
```

6.3 Dynamic Computational Graph in PyTorch

Computational Graphs allow a deep learning framework to do additional bookkeeping to implement automatic gradient differentiation needed to obtain gradients of parameters during training. A computational graph is a DAG (directed acyclic graph), where the nodes are the mathematical operations, such as multiplication and addition. The inputs to the operations are the incoming edges and the output of each operation is the outgoing edge.

Static frameworks such as Teano, Caffe and TensorFlow (< v1.7) require the computational graphs to be first declared , compiled and the excuted. Although this leads to extremely efficient implementations (useful in production and mobile settings), it can become quite cumberson during research and development. Modern frames like Chainer, DyNet and Pytorch, implement Dynamic Computational Graphs to allow for a more flexible, imperative style of development, without needing to compile the models before every excution. Dynamic computational graphs are especially useful in modelling NLP tasks as each input could potentially result in a different graph structure. As of Version 1.7, TensorFlow has an eager excution that `tf.Tensor` objects reference concrete values instead of symbolic handles to nodes in a computational graph. This is a similar to the PyTorch's dynamic computational graph idea.

```
import torch
import numpy as np

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

6.3.1 Training Data Preparation

We use the same linear model data preparation process here.

```
true_b = 1
true_w = 2
N = 100
# Data Generation
np.random.seed(42)
x = np.random.rand(N, 1)
epsilon = (.1 * np.random.randn(N, 1))
y = true_b + true_w * x + epsilon

# Shuffles the indices
idx = np.arange(N)
np.random.shuffle(idx)
# Uses first 80 random indices for train
train_idx = idx[:int(N*.8)]
# Uses the remaining indices for validation
val_idx = idx[int(N*.8):]
# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

6.3.2 Using `torchviz` to visualise computation graph

```
from torch import nn
from torchviz import make_dot, make_dot_from_trace
```

```
# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
    dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
    dtype=torch.float, device=device)
# Step 1 - Computes our model's predicted output
#        - forward pass
yhat = b + w * x_train_tensor
# Step 2 - Computes the loss
error = (yhat - y_train_tensor)
loss = (error ** 2).mean()
# We can try plotting the graph for any python variable:
# yhat, error, loss...
make_dot(yhat)
```

```
<graphviz.dot.Digraph at 0x1cd4650f580>
```

Let's take a closer look at its components:

- blue boxes ((1)s): these boxes correspond to the tensors we use as parameters, the ones we're asking PyTorch to compute gradients for
- gray boxes (MulBackward0 and AddBackward0): a Python operation that involves a gradient-computing tensor or its dependencies
- green box ((80, 1)): the tensor used as the starting point for the computation of gradients (assuming the backward() method is called from the variable used to visualize the graph) — they are computed from the bottom-up in a graph

Now, take a closer look at the gray box (AddBackward0) at the bottom of the graph: two arrows are pointing to it since it is adding up two variables, b, and $w \times x$. But the gray box (MulBackward0) of the same graph: it is performing a multiplication, namely, $w \times x$. But there is only one arrow pointing to it! The arrow comes from the blue box that corresponds to our parameter w.

"Why don't we have a box for our data (x)?"

The answer is: we do not compute gradients for it!

6.3.3 No gradient tracking for `b`

Important: Even though there are more tensors involved in the operations performed by the computation graph, it only shows **gradient-computing tensors** and its dependencies.

Let's try to set b to be a non-gradient computing tensor and visualise the computational graph.

```
b_nograd = torch.randn(1, requires_grad=False, \
dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
```

(continues on next page)

(continued from previous page)

```
dtype=torch.float, device=device)
yhat = b_nograd + w * x_train_tensor
make_dot(yhat)
```

```
<graphviz.dot.Digraph at 0x1cd4c14b910>
```

6.3.4 Computational Graph with branches

Let's look at how computational graph deals with if statement. Note the code does not do anything sensible, purely for visualisation and demonstration purposes.

```
b = torch.randn(1, requires_grad=True, \
dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
dtype=torch.float, device=device)
yhat = b + w * x_train_tensor
error = yhat - y_train_tensor
loss = (error ** 2).mean()
# this makes no sense!!
if loss > 0:
    yhat2 = w * x_train_tensor
    error2 = yhat2 - y_train_tensor
# neither does this :-)
loss += error2.mean()
make_dot(loss)
```

```
<graphviz.dot.Digraph at 0x1cd4c17b1c0>
```

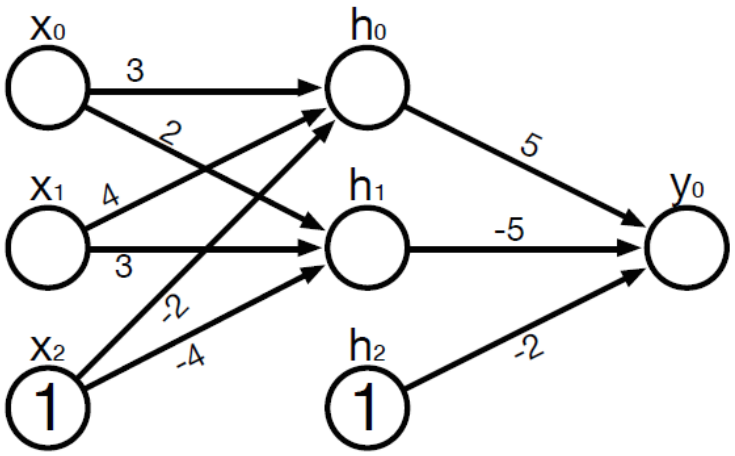
6.4 Neural Networks in PyTorch

6.4.1 A neural network with one hidden layer

Extending the simple perceptron in the openning page of this lab, let's build a simple neural network that takes two binary inputs, and simulate the logical operation of XOR. The network has two sets of weights and biases, one set between the input and the hidden layer with two nodes, another set between the hidden layer and the single output, as shown in the figure below.

Note: Compare the code below with the `Perceptron` code at the front page of this lab to have a better understanding of the building blocks.

```
import torch
import torch.nn as nn
import torch.optim as optim
```



Defining the model class

Let's define the XOR neural network model inherited from the `torch.nn.Module` class using an Object Oriented approach.

```
class XOR(nn.Module):
    """
    An XOR is similuated using neural network with
    two fully connected linear layers
    """

    def __init__(self, input_dim, output_dim):
        """
        Args:
            input_dim (int): size of the input features
            output_dim (int): size of the output
        """
        super(XOR, self).__init__()
        self.fc1 = nn.Linear(input_dim, 2)
        self.fc2 = nn.Linear(2, output_dim)

    def forward(self, x_in):
        """The forward pass of the perceptron

        Args:
            x_in (torch.Tensor): an input data tensor
                x_in.shape should be (batch, num_features)
        Returns:
            the resulting tensor. tensor.shape should be (batch,).
        """
        hidden = torch.sigmoid(self.fc1(x_in))
        yhat = torch.sigmoid(self.fc2(hidden))
        return yhat
```

Object Orientation in PyTorch

In PyTorch, a model (e.g. the `XOR` model) is represented by a regular Python class that inherits from the Module class.

Important: IMPORTANT: If you are uncomfortable with object-oriented programming (OOP) concepts like *classes*, *constructors*, *methods/class methods*, *instances*, and *attributes*, it is strongly recommended to follow tutorials such as Real Python's Objected-Oriented Programming (OOP) in Python 3

The most fundamental methods a model class needs to implement are:

- `__init__(self)`: it defines the parts that make up the model — in our case, two parameters, b and w.
- `forward(self, x)`: it performs the actual computation, that is, it outputs a prediction, given the input x.

Note: Do not forget to include `super().__init__()` to execute the `__init__()` method of the parent class `(nn.Module)` before your own.

6.4.2 XOR Model

Now let's create a Linear Model with two inputs (features) and one output.

Calling the XOR constructor with an `input_dim=2` and `output_dim=1`, namely `XOR(2,1)` results in two fully connected linear models, one `nn.Linear(2,2)` and ``nn.Linear(2,1)`, which will create a model with two input features, one output feature with biases at the input layer, hidden layer and output layer.

```
model = XOR(2,1)
```

Obtain all model parameters using `state_dict()`

We can get the current values of all parameters using our model's state_dict() method.

```
model.state_dict()
```

```
OrderedDict([('fc1.weight',
              tensor([[ 0.5827,  0.0717],
                      [-0.2969, -0.1214]])),
             ('fc1.bias', tensor([-0.5186, -0.6406])),
             ('fc2.weight', tensor([[0.3564, 0.5904]])),
             ('fc2.bias', tensor([0.3660]))])
```

We used to manually assign random values to these weights and biases. Now PyTorch does it for us automatically.

The `state_dict()` of a given model is simply a Python dictionary that maps each attribute/parameter to its corresponding tensor. But only learnable parameters are included, as its purpose is to keep track of parameters that are going to be updated by the optimizer.

The optimizer itself has a `state_dict()` too, which contains its internal state, as well as other hyper-parameters. Let's take a quick look at it:

```
lr = 0.01
optimizer = optim.SGD(model.parameters(), lr=lr)
optimizer.state_dict()
```

```
{'state': {},
 'param_groups': [{'lr': 0.01,
   'momentum': 0,
   'dampening': 0,
   'weight_decay': 0,
   'nesterov': False,
   'params': [0, 1, 2, 3]}]}
```

Model and Data need to be on the same device

Important: We need to send our model to the same device where the data is. If our data is made of GPU tensors, our model must "live" inside the GPU as well.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
model = model.to(device)
```

6.4.3 Training XOR

Now let's put it all together to train an neural XOR model.

Training Data Preparation

```
# Training data preparation

x_train_tensor = torch.tensor([[0,0],[0,1],[1,1],[1,0]], device=device).float()
y_train_tensor = torch.tensor([0,1,1,0], device=device).view(4,1).float()

x_val_tensor = torch.clone(x_train_tensor)
y_val_tensor = torch.clone(y_train_tensor)
```

```
# Verify the shape of the output tensor
y_train_tensor.shape
```

```
torch.Size([4, 1])
```

Hyperparameter setup

We need to set up the learning rate and the number of epochs, and then select the three key compoenents of a neural model: model, optimiser and loss function before training.

```
# Sets learning rate - this is "eta" ~ the "n" like
# Greek letter
lr = 0.01

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
```

(continues on next page)

(continued from previous page)

```
# Now we can create a model and send it at once to the device
model = XOR(2,1)
model = model.to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Defines number of epochs
n_epochs = 100000
```

Training

Now we are ready to train.

```
for epoch in range(n_epochs):
    #for j in range(steps):
    model.train() # What is this?!?

    # Step 1 - Computes model's predicted output - forward pass
    # No more manual prediction!
    yhat = model(x_train_tensor)

    # Step 2 - Computes the loss
    loss = loss_fn(yhat, y_train_tensor)

    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()
    if (epoch % 500 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(epoch, loss.to("cpu").detach().
 ↪numpy()))

# We can also inspect its parameters using its state_dict
print(model.state_dict())
```

```
Epoch: 0, Loss: 0.27375736832618713,
Epoch: 500, Loss: 0.2555079460144043,
Epoch: 1000, Loss: 0.25011080503463745,
Epoch: 1500, Loss: 0.24655020236968994,
Epoch: 2000, Loss: 0.24295492470264435,
Epoch: 2500, Loss: 0.238966703414917,
Epoch: 3000, Loss: 0.23448437452316284,
Epoch: 3500, Loss: 0.22930260002613068,
Epoch: 4000, Loss: 0.2232476770877838,
Epoch: 4500, Loss: 0.21623794734477997,
Epoch: 5000, Loss: 0.20812170207500458,
Epoch: 5500, Loss: 0.19883160293102264,
```

(continues on next page)

(continued from previous page)

```
Epoch: 6000, Loss: 0.18850544095039368,
Epoch: 6500, Loss: 0.1771014928817749,
Epoch: 7000, Loss: 0.16488364338874817,
Epoch: 7500, Loss: 0.15211743116378784,
Epoch: 8000, Loss: 0.139143705368042,
Epoch: 8500, Loss: 0.12626223266124725,
Epoch: 9000, Loss: 0.11392521113157272,
Epoch: 9500, Loss: 0.10234947502613068,
Epoch: 10000, Loss: 0.09172774851322174,
Epoch: 10500, Loss: 0.08199518918991089,
Epoch: 11000, Loss: 0.0733182281255722,
Epoch: 11500, Loss: 0.06560301780700684,
Epoch: 12000, Loss: 0.05884382501244545,
Epoch: 12500, Loss: 0.05294763296842575,
Epoch: 13000, Loss: 0.04782135412096977,
Epoch: 13500, Loss: 0.04328809678554535,
Epoch: 14000, Loss: 0.039366669952869415,
Epoch: 14500, Loss: 0.03590844199061394,
Epoch: 15000, Loss: 0.03286368399858475,
Epoch: 15500, Loss: 0.030230185016989708,
Epoch: 16000, Loss: 0.02787426859140396,
Epoch: 16500, Loss: 0.025831308215856552,
Epoch: 17000, Loss: 0.02391224540770054,
Epoch: 17500, Loss: 0.02228483371436596,
Epoch: 18000, Loss: 0.02080184407532215,
Epoch: 18500, Loss: 0.019479243084788322,
Epoch: 19000, Loss: 0.018279727548360825,
Epoch: 19500, Loss: 0.017226679250597954,
Epoch: 20000, Loss: 0.01625426858663559,
Epoch: 20500, Loss: 0.01537842396646738,
Epoch: 21000, Loss: 0.014549647457897663,
Epoch: 21500, Loss: 0.01378196757286787,
Epoch: 22000, Loss: 0.01309845969080925,
Epoch: 22500, Loss: 0.01248301099985838,
Epoch: 23000, Loss: 0.011914841830730438,
Epoch: 23500, Loss: 0.011366013437509537,
Epoch: 24000, Loss: 0.01088915579020977,
Epoch: 24500, Loss: 0.010406192392110825,
Epoch: 25000, Loss: 0.010014466941356659,
Epoch: 25500, Loss: 0.009598497301340103,
Epoch: 26000, Loss: 0.009224366396665573,
Epoch: 26500, Loss: 0.008882050402462482,
Epoch: 27000, Loss: 0.008561154827475548,
Epoch: 27500, Loss: 0.008238466456532478,
Epoch: 28000, Loss: 0.007974416017532349,
Epoch: 28500, Loss: 0.007680158130824566,
Epoch: 29000, Loss: 0.007433216087520l225,
Epoch: 29500, Loss: 0.007197014521807432,
Epoch: 30000, Loss: 0.006972037721425295,
Epoch: 30500, Loss: 0.0067596533335745335,
Epoch: 31000, Loss: 0.006562951020896435,
Epoch: 31500, Loss: 0.006372722331434488,
Epoch: 32000, Loss: 0.006173261906951666,
Epoch: 32500, Loss: 0.005995258688926697,
Epoch: 33000, Loss: 0.005834578536450863,
Epoch: 33500, Loss: 0.005697771906852722,
Epoch: 34000, Loss: 0.005533096380531788,
```

(continues on next page)

(continued from previous page)

```
Epoch: 34500, Loss: 0.005399664863944054,
Epoch: 35000, Loss: 0.005252258852124214,
Epoch: 35500, Loss: 0.0051279375329613686,
Epoch: 36000, Loss: 0.005004711449146271,
Epoch: 36500, Loss: 0.00487515376880765,
Epoch: 37000, Loss: 0.004761365242302418,
Epoch: 37500, Loss: 0.004653751850128174,
Epoch: 38000, Loss: 0.004538621753454208,
Epoch: 38500, Loss: 0.004451357759535313,
Epoch: 39000, Loss: 0.004348565824329853,
Epoch: 39500, Loss: 0.004250797443091869,
Epoch: 40000, Loss: 0.0041722762398421764,
Epoch: 40500, Loss: 0.00408073328435421,
Epoch: 41000, Loss: 0.004001058172434568,
Epoch: 41500, Loss: 0.003916418645530939,
Epoch: 42000, Loss: 0.0038418788462877274,
Epoch: 42500, Loss: 0.0037747276946902275,
Epoch: 43000, Loss: 0.003688385244458914,
Epoch: 43500, Loss: 0.003622728865593672,
Epoch: 44000, Loss: 0.0035602175630629063,
Epoch: 44500, Loss: 0.003489775350317359,
Epoch: 45000, Loss: 0.003426765091717243,
Epoch: 45500, Loss: 0.003366705495864153,
Epoch: 46000, Loss: 0.0033056712709367275,
Epoch: 46500, Loss: 0.0032519223168492317,
Epoch: 47000, Loss: 0.0031912513077259064,
Epoch: 47500, Loss: 0.0031480954494327307,
Epoch: 48000, Loss: 0.003087468910962343,
Epoch: 48500, Loss: 0.0030377130024135113,
Epoch: 49000, Loss: 0.0029869868885725737,
Epoch: 49500, Loss: 0.0029457740020006895,
Epoch: 50000, Loss: 0.0028982609510421753,
Epoch: 50500, Loss: 0.0028544270899146795,
Epoch: 51000, Loss: 0.0028089506085962057,
Epoch: 51500, Loss: 0.0027682341169565916,
Epoch: 52000, Loss: 0.0027215657755732536,
Epoch: 52500, Loss: 0.0026850481517612934,
Epoch: 53000, Loss: 0.002641090890392661,
Epoch: 53500, Loss: 0.0026040119118988514,
Epoch: 54000, Loss: 0.0025728303007781506,
Epoch: 54500, Loss: 0.0025344102177768946,
Epoch: 55000, Loss: 0.0024971726816147566,
Epoch: 55500, Loss: 0.002462733769789338,
Epoch: 56000, Loss: 0.002433920744806528,
Epoch: 56500, Loss: 0.0023974804207682261,
Epoch: 57000, Loss: 0.0023658026475459337,
Epoch: 57500, Loss: 0.002332453615963459,
Epoch: 58000, Loss: 0.0023027895949780094,
Epoch: 58500, Loss: 0.0022734100930392741,
Epoch: 59000, Loss: 0.0022432627156376841,
Epoch: 59500, Loss: 0.0022170250304043293,
Epoch: 60000, Loss: 0.0021900190046046917,
Epoch: 60500, Loss: 0.0021582804620265961,
Epoch: 61000, Loss: 0.0021350218448787928,
Epoch: 61500, Loss: 0.0021104959305375814,
Epoch: 62000, Loss: 0.0020827956032007933,
Epoch: 62500, Loss: 0.002065179403871298,
```

(continues on next page)

(continued from previous page)

```
Epoch: 63000, Loss: 0.002039001788944006,
Epoch: 63500, Loss: 0.0020126295275986195,
Epoch: 64000, Loss: 0.0019906111992895603,
Epoch: 64500, Loss: 0.0019670012407004833,
Epoch: 65000, Loss: 0.0019415951101109385,
Epoch: 65500, Loss: 0.0019239818211644888,
Epoch: 66000, Loss: 0.0019025187939405441,
Epoch: 66500, Loss: 0.001879572868347168,
Epoch: 67000, Loss: 0.0018573695560917258,
Epoch: 67500, Loss: 0.0018442103173583746,
Epoch: 68000, Loss: 0.0018211827846243978,
Epoch: 68500, Loss: 0.0017985039157792926,
Epoch: 69000, Loss: 0.0017842620145529509,
Epoch: 69500, Loss: 0.001763233682140708,
Epoch: 70000, Loss: 0.0017411105800420046,
Epoch: 70500, Loss: 0.0017304448410868645,
Epoch: 71000, Loss: 0.0017097863601520658,
Epoch: 71500, Loss: 0.001695991144515574,
Epoch: 72000, Loss: 0.0016756795812398195,
Epoch: 72500, Loss: 0.0016603447729721665,
Epoch: 73000, Loss: 0.0016442921478301287,
Epoch: 73500, Loss: 0.0016290463972836733,
Epoch: 74000, Loss: 0.0016133070457726717,
Epoch: 74500, Loss: 0.0015992799308151007,
Epoch: 75000, Loss: 0.001579604228027165,
Epoch: 75500, Loss: 0.0015679539646953344,
Epoch: 76000, Loss: 0.0015530944801867008,
Epoch: 76500, Loss: 0.0015387125313282013,
Epoch: 77000, Loss: 0.0015230522258207202,
Epoch: 77500, Loss: 0.0015115310670807958,
Epoch: 78000, Loss: 0.0014988419134169817,
Epoch: 78500, Loss: 0.0014816472539678216,
Epoch: 79000, Loss: 0.0014696801081299782,
Epoch: 79500, Loss: 0.00145495415199548,
Epoch: 80000, Loss: 0.0014449709560722113,
Epoch: 80500, Loss: 0.001434221281670034,
Epoch: 81000, Loss: 0.0014170538634061813,
Epoch: 81500, Loss: 0.0014048947487026453,
Epoch: 82000, Loss: 0.001396734151057899,
Epoch: 82500, Loss: 0.0013808537041768432,
Epoch: 83000, Loss: 0.0013706223107874393,
Epoch: 83500, Loss: 0.001362814218737185,
Epoch: 84000, Loss: 0.0013511404395103455,
Epoch: 84500, Loss: 0.0013351887464523315,
Epoch: 85000, Loss: 0.0013270438648760319,
Epoch: 85500, Loss: 0.0013187713921070099,
Epoch: 86000, Loss: 0.0013084793463349342,
Epoch: 86500, Loss: 0.0012941481545567513,
Epoch: 87000, Loss: 0.0012852461077272892,
Epoch: 87500, Loss: 0.0012746803695335984,
Epoch: 88000, Loss: 0.0012681942898780107,
Epoch: 88500, Loss: 0.0012594859581440687,
Epoch: 89000, Loss: 0.0012420803541317582,
Epoch: 89500, Loss: 0.0012359283864498138,
Epoch: 90000, Loss: 0.0012270398437976837,
Epoch: 90500, Loss: 0.001218495424836874,
Epoch: 91000, Loss: 0.0012108510127291083,
```

(continues on next page)

(continued from previous page)

```
Epoch: 91500, Loss: 0.0012005458120256662,
Epoch: 92000, Loss: 0.001190779497846961,
Epoch: 92500, Loss: 0.001179547980427742,
Epoch: 93000, Loss: 0.001171827781945467,
Epoch: 93500, Loss: 0.001165188499726355,
Epoch: 94000, Loss: 0.0011571240611374378,
Epoch: 94500, Loss: 0.001148390700109303,
Epoch: 95000, Loss: 0.0011400426737964153,
Epoch: 95500, Loss: 0.001132698031142354,
Epoch: 96000, Loss: 0.0011260239407420158,
Epoch: 96500, Loss: 0.0011176535626873374,
Epoch: 97000, Loss: 0.0011106508318334818,
Epoch: 97500, Loss: 0.0011001934763044119,
Epoch: 98000, Loss: 0.0010945061221718788,
Epoch: 98500, Loss: 0.0010835299035534263,
Epoch: 99000, Loss: 0.001077619381248951,
Epoch: 99500, Loss: 0.0010701098944991827,
OrderedDict([('fc1.weight', tensor([[ 0.2956, -2.3485],
        [-0.1467,  4.8632]], device='cuda:0')), ('fc1.bias', tensor([ 0.7222, -2.
↪1771], device='cuda:0')), ('fc2.weight', tensor([[-2.9656,  6.3072]], device='cuda:0
↪')), ('fc2.bias', tensor([-1.8895], device='cuda:0'))])
```

Important: In PyTorch, models have a train() method which, somewhat disappointingly, does NOT perform a training step. Its only purpose is to set the model to **training mode**. Why is this important? Some models may use mechanisms like Dropout, for instance, which have distinct behaviors during training and evaluation phases.

It is good practice to call `model.train()` in the training loop. It is also possible to set a model to evaluation mode. We will see this in later labs.

Your Turn

Put the returned weights and biases into the XOR neural network diagram and try to work out the output when the input is `[0,0]`.

Inference (Forward Pass)

Instead of verifying it maually, we can test out our XOR model, with input [0,1] by called the model with the input. Note we do not call the forward function directly, instead we provide input to the model.

```
model(torch.tensor([0.,1.]).to(device))
```

```
tensor([0.9715], device='cuda:0', grad_fn=<SigmoidBackward>)
```

6.4.4 Logging the model training for Visualisation in TensorBoard

TensorBoard by TensorFlow is a very useful tool for visualising training progress and model architectures, despite being a competiting platform, PyTorch provides classes and methods for us to integrate it with our model.

TensorBoard can be loaded inside Jupyter notebooks, or can be started externally from a command line. Before we run TensorBoard, we need to change to the directory of your model code, and create a folder, e.g. `runs` to keep a log of the training progress.

The examples in this lab are running Tensorboard inside a notebook, but it is a good idea to run TensorBoard on the command line by giving the log directory. Assuming you are one level up the `runs` directory:

```
tensorboard --logdir runs
```

If successful, it will say that `TensorBoard 2.6.0 at http://localhost:6006/ (Press CTRL+C to quit)`, copy and paste the URL to your browser to see TensorBoard in action.

```
%load_ext tensorboard
```

TensorBoard running in Jupyter Notebook

```
%tensorboard --logdir runs
```

SummaryWriter

It all starts with the creation of a SummaryWriter. TensorBoard to look for logs inside the `runs` folder, it only makes sense to actually log to that folder. Moreover, to be able to distinguish between different experiments or models, we should also specify a sub-folder: test.

If we do not specify any folder, TensorBoard will default to `runs/CURRENT_DATETIME_HOSTNAME`, which is not such a great name if you'd be looking for your experiment results in the future.

So, it is recommended to try to name it in a more meaningful way, like runs/test or runs/simple_linear_regression. It will then create a subfolder inside runs (the folder we specified when we started TensorBoard).

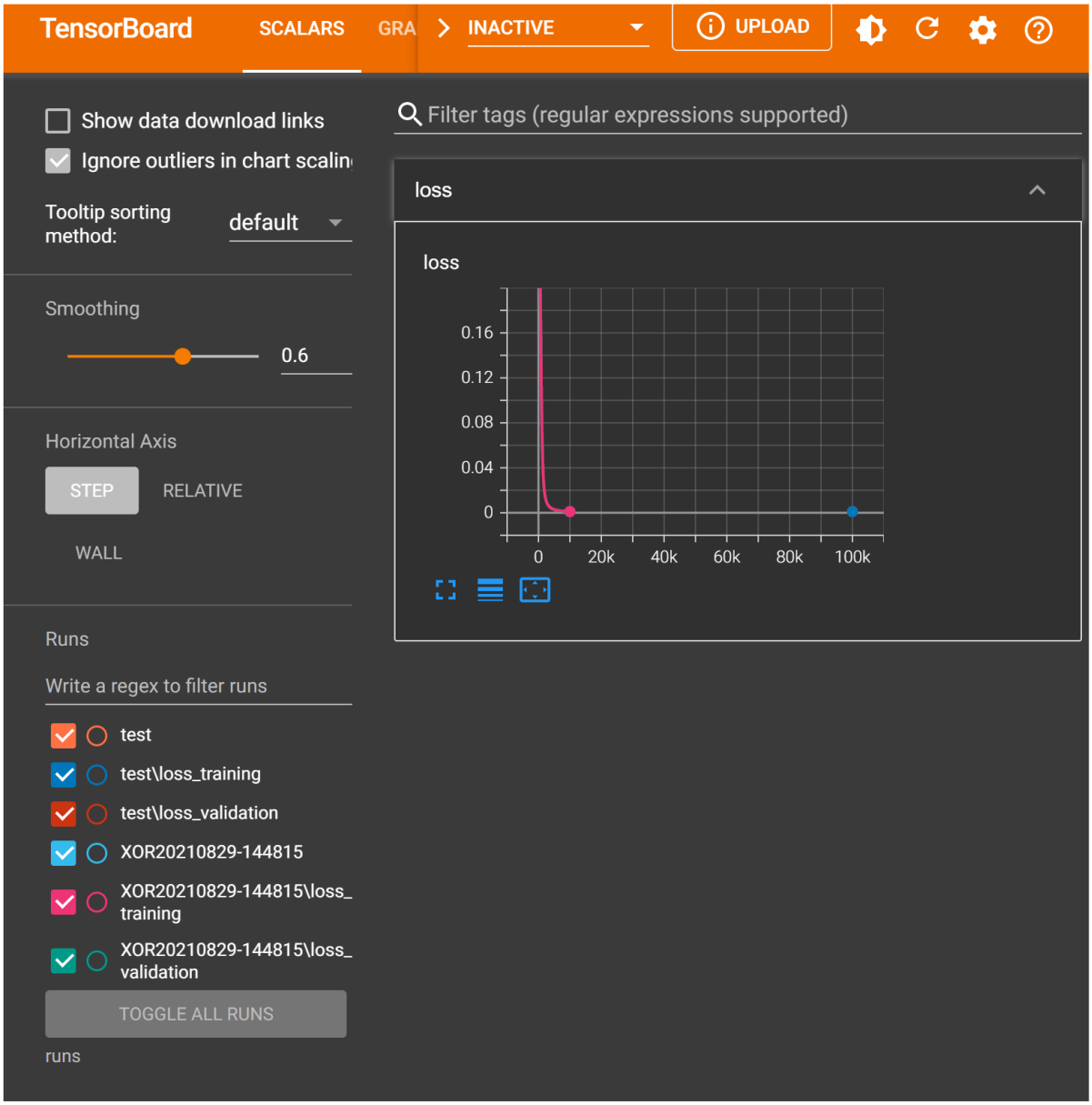
Even better, you should name it in a meaningful way and add datetime or a sequential number as a suffix, like `runs/test_001` or `runs/test_20200502172130`, to avoid writing data of multiple runs into the same folder (we'll see why this is bad in the add_scalars section below).

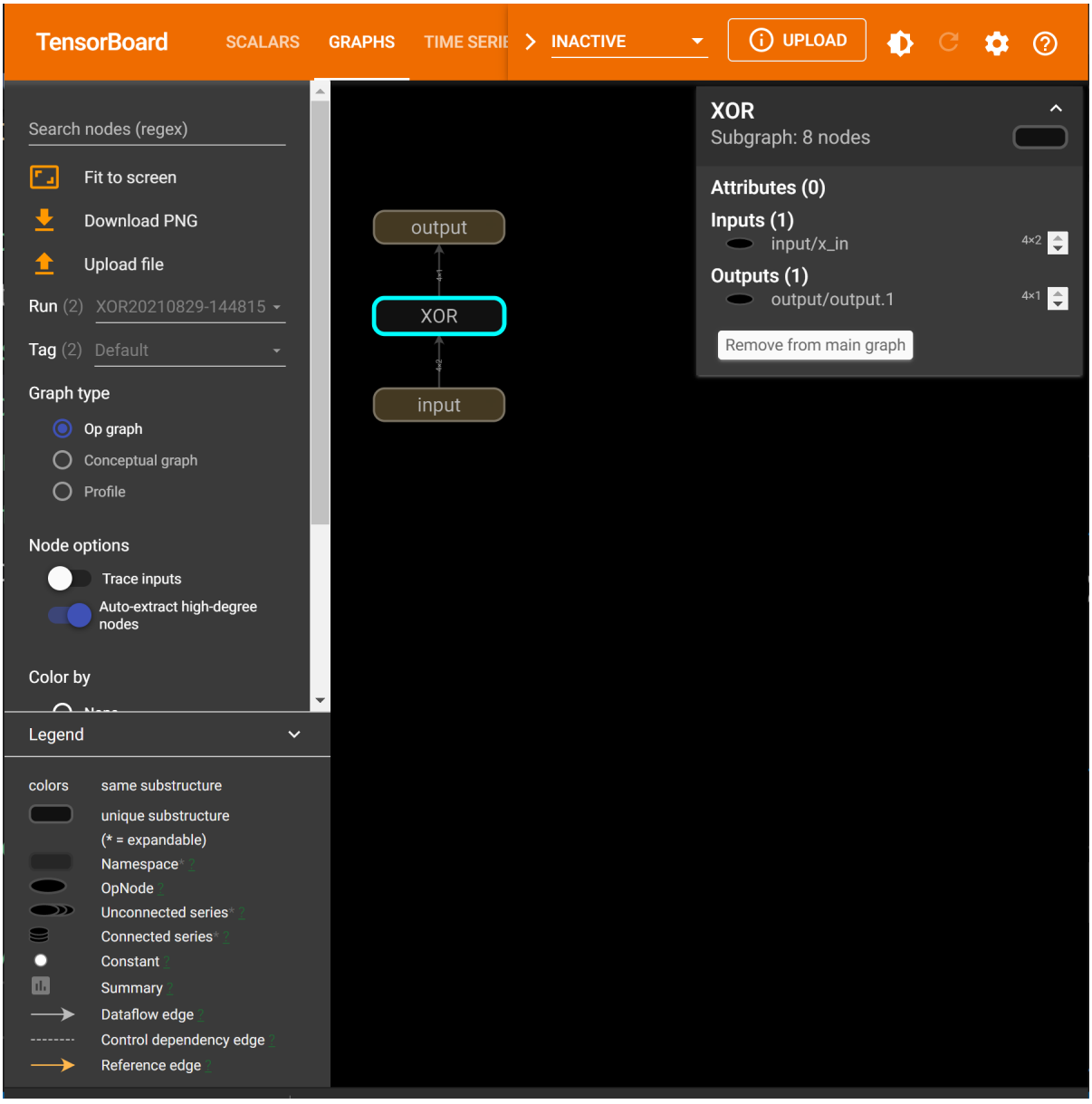
```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter('runs/test')
```

add_graph

It will produce an input-output graph that allows you to interactively inspect parameters, which is different from the TorchViz's computation graph (a static visualisation - not interactive).

```
writer.add_graph(model, x_train_tensor)
%tensorboard --logdir runs
```


TensorBoard
SCALARS
GRA
INACTIVE
UPLOAD
Show data download links
Ignore outliers in chart scalin
Tooltip sorting method:
default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
test
test\loss_training
test\loss_validation
XOR20210829-144815
XOR20210829-144815\loss_training
XOR20210829-144815\loss_validation
TOGGLE ALL RUNS
runs
Filter tags (regular expressions supported)
loss
loss
0.16
0.12
0.08
0.04
0
0
20k
40k
60k
80k
100k


TensorBoard
SCALARS
GRAPHS
TIME SERIE
INACTIVE
UPLOAD
Search nodes (regex)
Fit to screen
Download PNG
Upload file
Run (2)
XOR20210829-144815
Tag (2)
Default
Graph type
Op graph
Conceptual graph
Profile
Node options
Trace inputs
Auto-extract high-degree nodes
Color by
Legend
colors
same substructure
unique substructure
(* = expandable)
Namespace*
OpNode
Unconnected series*
Connected series*
Constant
Summary
Dataflow edge
Control dependency edge
Reference edge
output
XOR
input
XOR
Subgraph: 8 nodes
Attributes (0)
Inputs (1)
input/x_in
4×2
Outputs (1)
output/output.1
4×1
Remove from main graph

add_scalar

We can send the loss values to TensorBoard using the `add_scalars` method to send multiple scalar values at once, and it needs three arguments:

- main_tag: the parent name of the tags or, the "group tag"
- tag_scalar_dict: the dictionary containing the key: value pairs for the scalars you want to keep track of (can be training and validation losses)
- global_step: step value, that is, the index you're associating with the values you're sending in the dictionary - the epoch comes to mind in our case, as losses are computed for each epoch

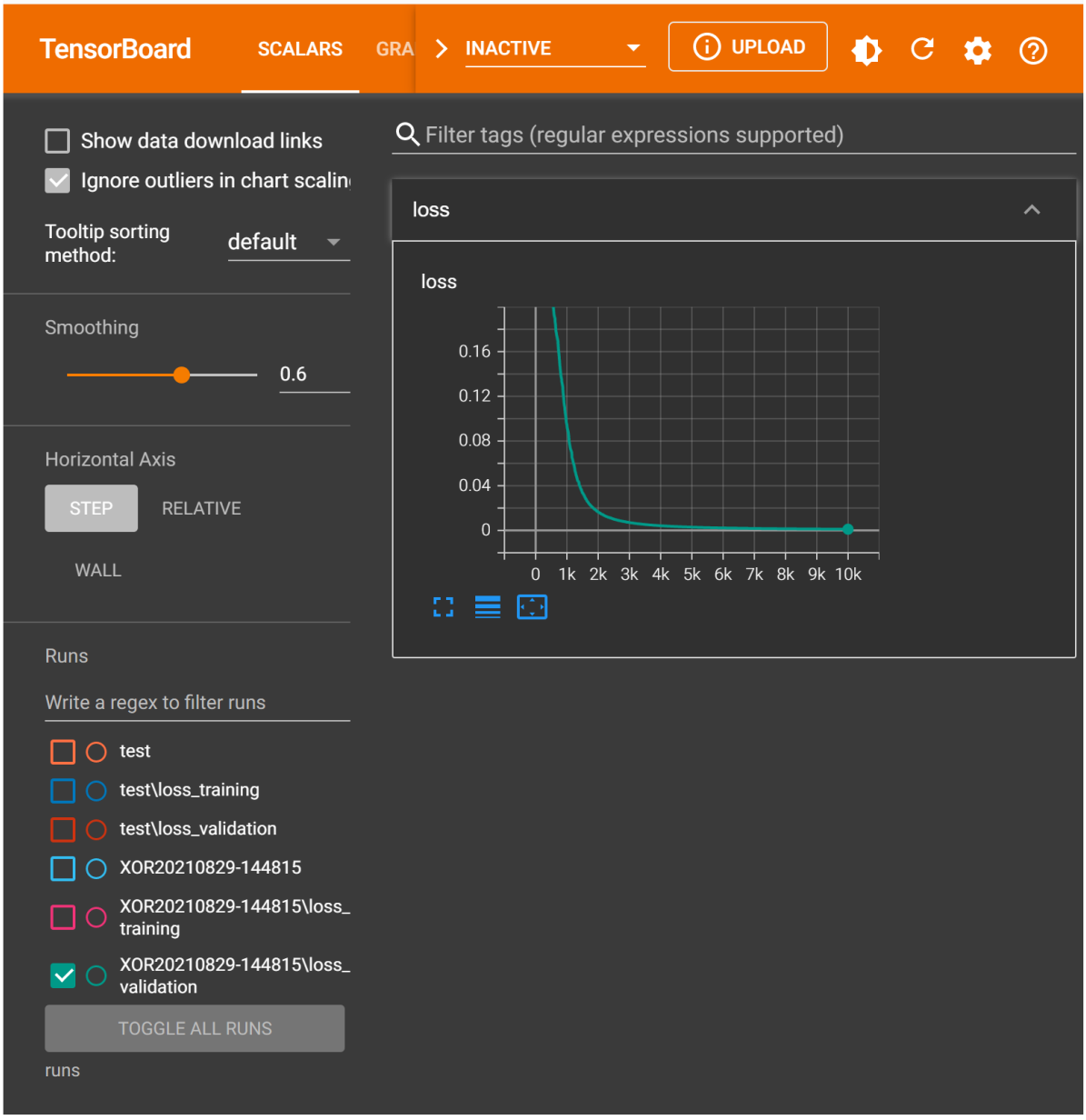
```
writer.add_scalars(main_tag='loss',
    tag_scalar_dict={'training': loss,
                     'validation': loss},
    global_step=epoch
)
```

If you run the code above after performing the model training, it will just send both loss values computed for the last epoch.

```
%tensorboard --logdir runs
```

```
from datetime import datetime
# Sets learning rate - this is "eta" ~ the "n" like
# Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = XOR(2,1)
model = model.to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Tensorboard setup
writer = SummaryWriter('runs/XOR' + datetime.now().strftime("%Y%m%d-%H%M%S"))
writer.add_graph(model, x_train_tensor.to(device))

# Defines number of epochs
n_epochs = 10000

losses = []
val_losses = [] # note we did not use the validation data
for epoch in range(n_epochs):
    #for j in range(steps):
    model.train() # What is this?!?

    # Step 1 - Computes model's predicted output - forward pass
    # No more manual prediction!
    yhat = model(x_train_tensor)
```

(continues on next page)


TensorBoard
SCALARS
GRA
INACTIVE
UPLOAD
Show data download links
Ignore outliers in chart scalin
Tooltip sorting method:
default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
test
test\loss_training
test\loss_validation
XOR20210829-144815
XOR20210829-144815\loss_training
XOR20210829-144815\loss_validation
TOGGLE ALL RUNS
runs
Filter tags (regular expressions supported)
loss
loss
0.16
0.12
0.08
0.04
0
0 1k 2k 3k 4k 5k 6k 7k 8k 9k 10k

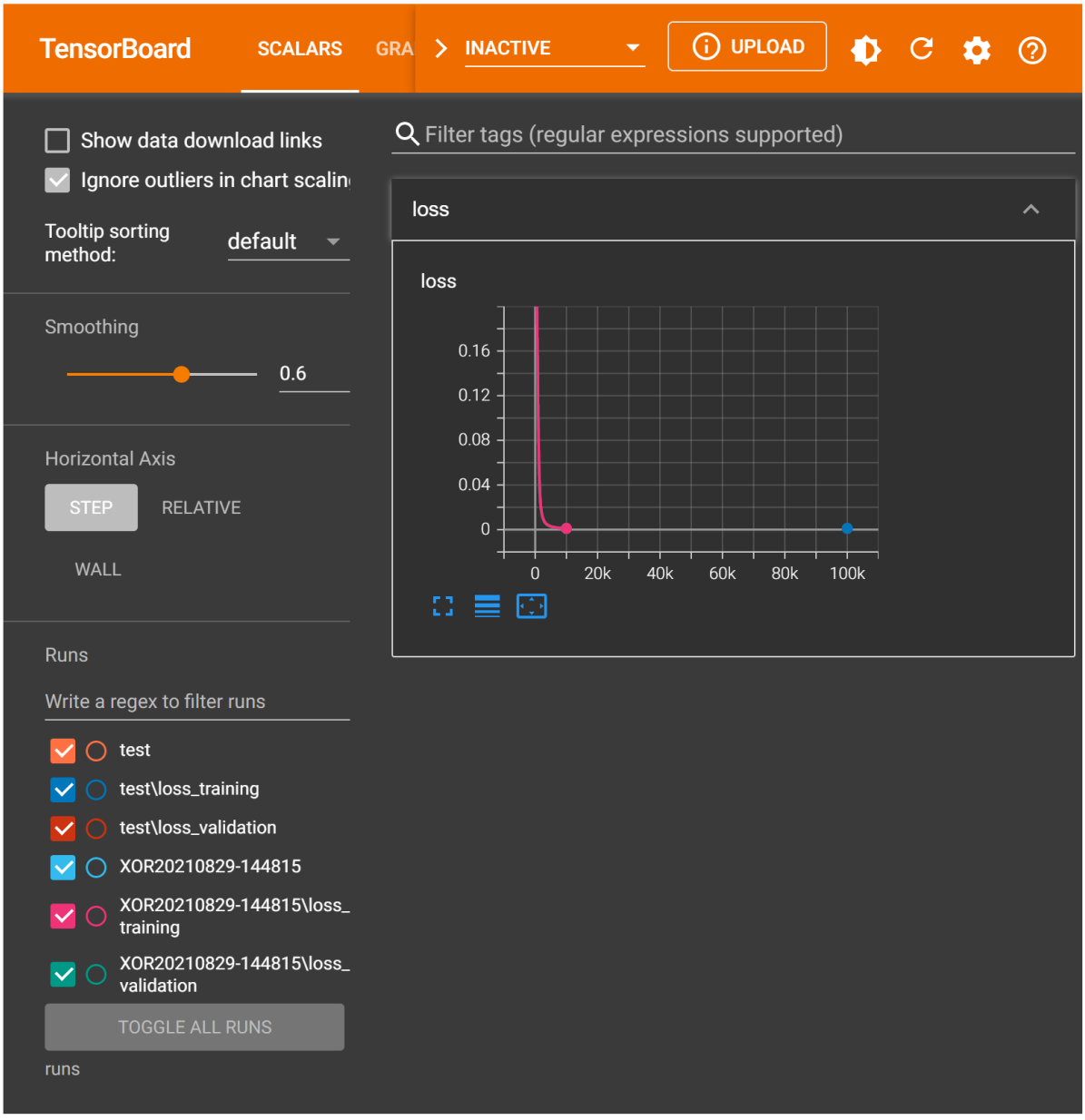
(continued from previous page)

```
    # Step 2 - Computes the loss
    loss = loss_fn(yhat, y_train_tensor)

    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()
    if (epoch % 500 == 0):
        print("Epoch: {0}, Loss: {1}, ".
            format(epoch, loss.to("cpu").detach().numpy()))

    losses.append(loss)
    writer.add_scalars(main_tag='loss',
                       tag_scalar_dict={'training': loss,
                                        'validation': loss},
                       global_step=epoch)

writer.close()
```

```
Epoch: 0, Loss: 0.27375736832618713,
Epoch: 500, Loss: 0.2081705629825592,
Epoch: 1000, Loss: 0.09176724404096603,
Epoch: 1500, Loss: 0.03289078176021576,
Epoch: 2000, Loss: 0.016263967379927635,
Epoch: 2500, Loss: 0.009998900815844536,
Epoch: 3000, Loss: 0.006978640332818031,
Epoch: 3500, Loss: 0.005261328537017107,
Epoch: 4000, Loss: 0.00416548689827323,
Epoch: 4500, Loss: 0.0034290249459445477,
Epoch: 5000, Loss: 0.002894176635891199,
Epoch: 5500, Loss: 0.002497166395187378,
Epoch: 6000, Loss: 0.0021936693228781223,
Epoch: 6500, Loss: 0.0019435270223766565,
Epoch: 7000, Loss: 0.0017444714903831482,
Epoch: 7500, Loss: 0.0015824229922145605,
Epoch: 8000, Loss: 0.001446553273126483,
Epoch: 8500, Loss: 0.0013299942947924137,
Epoch: 9000, Loss: 0.00122724543325603,
Epoch: 9500, Loss: 0.0011413537431508303,
```

```
%tensorboard --logdir runs
```

Note: In the TensorBoard logged run, we increased the learning rate and shortened the number of epochs. Play with these two parameters to see what you can get. Or change the loss function to BCE or BCEWithLogits to see how your training loss changes.


TensorBoard
SCALARS
GRA
INACTIVE
UPLOAD
Show data download links
Ignore outliers in chart scalin
Tooltip sorting method:
default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
test
test\loss_training
test\loss_validation
XOR20210829-144815
XOR20210829-144815\loss_training
XOR20210829-144815\loss_validation
TOGGLE ALL RUNS
runs
Filter tags (regular expressions supported)
loss
loss
0.16
0.12
0.08
0.04
0
0
20k
40k
60k
80k
100k

CHAPTER

SEVEN

LAB07: WORD EMBEDDINGS

Word Vectors are often used as a fundamental component for downstream NLP tasks, e.g. question answering, text generation, translation, etc., so it is important to build some intuitions as to their strengths and weaknesses. Here, you will explore two types of word vectors: those derived from *co-occurrence matrices*, and those derived via *GloVe*.

Note: The terms "word vectors" and "word embeddings" are often used interchangeably. The term "embedding" refers to the fact that we are encoding aspects of a word's meaning in a lower dimensional space. As Wikipedia states, "*conceptually it involves a mathematical embedding from a space with one dimension per word (one-hot encoding) to a continuous vector space with a much lower dimension*".

In this lab, we will look at how to use SVD on Word-Word co-ocurrence matrix to obtain word embeddings, and then visualising pretrained word embeddings using Matplotlib and TSNE, finally we train our own embeddings using a document crawled from the Web, and copmare with the pre-trained embeddings.

Credit: the notebooks are adapted from the Stanford CS224N Assignment on Exploring Word Vectors.

7.1 Environment Update with more packages

7.1.1 Install gensim

Note this will install the latest gensim 4.x

```
conda install -c conda-forge gensim
```

7.1.2 Install import-ipynb

This is a package to import functions from other Jupyter Notebooks.

```
pip install import-ipynb
```

7.1.3 Install Python Levenshtein Similarity

```
pip install python-Levenshtein
```

7.1.4 Install interactive visualisation bokeh

```
pip install bokeh
```

7.1.5 Install bs4 to get BeautifulSoup for web crawling

```
pip install bs4
```

7.2 Word Vectors from Word-Word Coocurrence Matrix

Let's take a look at how to use SVD on Word Coocurrence Matrix to obtain word vectors.

```
# All Import Statements Defined Here
# ----------------

import sys
assert sys.version_info[0]==3
assert sys.version_info[1] >= 5

from gensim.models import KeyedVectors
from gensim.test.utils import datapath
import pprint
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 5]
import nltk
nltk.download('reuters')
from nltk.corpus import reuters
import numpy as np
import random
import scipy as sp
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import PCA

START_TOKEN = '<START>'
END_TOKEN = '<END>'

np.random.seed(0)
random.seed(0)
# ----------------
```

```
C:\ProgramData\Anaconda3\envs\cits4012\lib\site-packages\gensim\similarities\__init__.
 ↪py:15: UserWarning: The gensim.similarities.levenshtein submodule is disabled,␣
 ↪because the optional Levenshtein package <https://pypi.org/project/python-
 ↪Levenshtein/> is unavailable. Install Levenhstein (e.g. `pip install python-
 ↪Levenshtein`) to suppress this warning.
  warnings.warn(msg)
```

(continues on next page)

(continued from previous page)

```
[nltk_data] Downloading package reuters to
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

Most word vector models start from the following idea:

You shall know a word by the company it keeps (Firth, J. R. 1957:11)

Many word vector implementations are driven by the idea that similar words, i.e., (near) synonyms, will be used in similar contexts. As a result, similar words will often be spoken or written along with a shared subset of words, i.e., contexts. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts. Here we elaborate upon one of those strategies, *co-occurrence matrices* (for more information, see here or here).

7.2.1 Co-Occurrence Matrix - Revisited

A co-occurrence matrix counts how often things co-occur in some environment. Given some word w_i occurring in the document, we consider the *context window* surrounding w_i. Supposing our fixed window size is n, then this is the n preceding and n subsequent words in that document, i.e. words $w_{i-n} \dots w_{i-1}$ and $w_{i+1} \dots w_{i+n}$. We build a *co-occurrence matrix* M, which is a symmetric word-by-word matrix in which M_{ij} is the number of times w_j appears inside w_i's window among all documents.

Example: Co-Occurrence with Fixed Window of n=1:

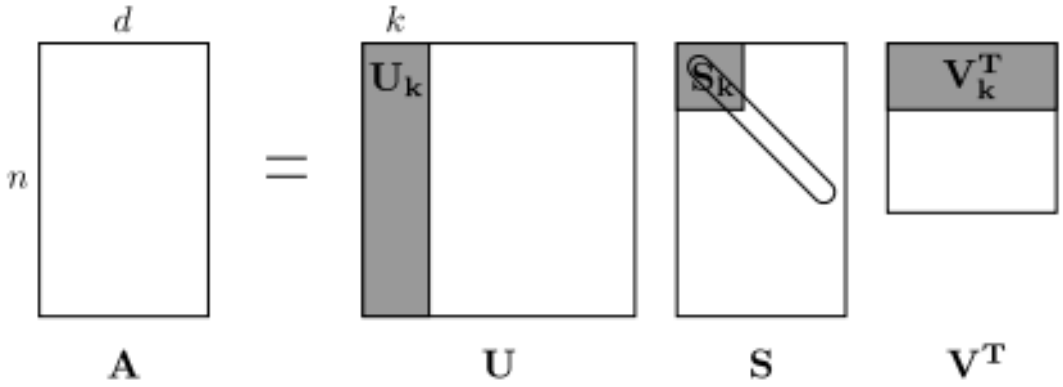
Document 1: "all that glitters is not gold"

Document 2: "all is well that ends well"

*	<START>	all	that	glitters	is	not	gold	well	ends	<END>
<START>	0	2	0	0	0	0	0	0	0	0
all	2	0	1	0	1	0	0	0	0	0
that	0	1	0	1	0	0	0	1	1	0
glitters	0	0	1	0	1	0	0	0	0	0
is	0	1	0	1	0	1	0	1	0	0
not	0	0	0	0	1	0	1	0	0	0
gold	0	0	0	0	0	1	0	0	0	1
well	0	0	1	0	1	0	0	0	1	1
ends	0	0	1	0	0	0	0	1	0	0
<END>	0	0	0	0	0	0	1	1	0	0

Note: In NLP, we often add <START> and <END> tokens to represent the beginning and end of sentences, paragraphs or documents. In thise case we imagine <START> and <END> tokens encapsulating each document, e.g., "<START> All that glitters is not gold <END>", and include these tokens in our co-occurrence counts.

The rows (or columns) of this matrix provide one type of word vectors (those based on word-word co-occurrence), but the vectors will be large in general (linear in the number of distinct words in a corpus). Thus, our next step is to run *dimensionality reduction*. In particular, we will run *SVD (Singular Value Decomposition)*, which is a kind of generalized *PCA (Principal Components Analysis)* to select the top k principal components. Here's a visualization of dimensionality reduction with SVD. In this picture our co-occurrence matrix is A with n rows corresponding to n words. We obtain a full matrix decomposition, with the singular values ordered in the diagonal S matrix, and our new, shorter length-k word vectors in U_k.



This reduced-dimensionality co-occurrence representation preserves semantic relationships between words, e.g. *doctor* and *hospital* will be closer than *doctor* and *dog*.

Note: If you can barely remember what an eigenvalue is, here's a slow, friendly introduction to SVD. If you want to learn more thoroughly about PCA or SVD, feel free to check out lectures 7, 8, and 9 of CS168. These course notes provide a great high-level treatment of these general purpose algorithms. Though, for the purpose of this class, you only need to know how to extract the k-dimensional embeddings by utilizing pre-programmed implementations of these algorithms from the numpy, scipy, or sklearn python packages. In practice, it is challenging to apply full SVD to large corpora because of the memory needed to perform PCA or SVD. However, if you only want the top k vector components for relatively small k — known as Truncated SVD — then there are reasonably scalable techniques to compute those iteratively.

7.2.2 Plotting Co-Occurrence Word Embeddings

Here, we will be using the Reuters (business and financial news) corpus. If you haven't run the import cell at the top of this page, please run it now (click it and press SHIFT-RETURN). The corpus consists of 10,788 news documents totaling 1.3 million words. These documents span 90 categories and are split into train and test. For more details, please see https://www.nltk.org/book/ch02.html. We provide a `read_corpus` function below that pulls out only articles from the "crude" (i.e. news articles about oil, gas, etc.) category. The function also adds `<START>` and `<END>` tokens to each of the documents, and lowercases words. You do **not** have to perform any other kind of pre-processing.

```
def read_corpus(category="crude"):
    """ Read files from the specified Reuter's category.
        Params:
            category (string): category name
        Return:
            list of lists, with words from each of the processed files
    """
    files = reuters.fileids(category)
    return [[START_TOKEN] + [w.lower() for w in list(reuters.words(f))] + [END_TOKEN]␣
↪for f in files]
```

Let's have a look what these documents are like….

```
reuters_corpus = read_corpus()
pprint.pprint(reuters_corpus[:3], compact=True, width=100)
```

```
[['<START>', 'japan', 'to', 'revise', 'long', '-', 'term', 'energy', 'demand',
↪'downwards', 'the',
 'ministry', 'of', 'international', 'trade', 'and', 'industry', '(', 'miti', ')',
↪'will', 'revise',
 'its', 'long', '-', 'term', 'energy', 'supply', '/', 'demand', 'outlook', 'by',
↪'august', 'to',
 'meet', 'a', 'forecast', 'downtrend', 'in', 'japanese', 'energy', 'demand', ',',
↪'ministry',
 'officials', 'said', '.', 'miti', 'is', 'expected', 'to', 'lower', 'the',
↪'projection', 'for',
 'primary', 'energy', 'supplies', 'in', 'the', 'year', '2000', 'to', '550', 'mln',
↪'kilolitres',
 '(', 'kl', ')', 'from', '600', 'mln', ',', 'they', 'said', '.', 'the', 'decision',
↪'follows',
 'the', 'emergence', 'of', 'structural', 'changes', 'in', 'japanese', 'industry',
↪'following',
 'the', 'rise', 'in', 'the', 'value', 'of', 'the', 'yen', 'and', 'a', 'decline', 'in
↪', 'domestic',
 'electric', 'power', 'demand', '.', 'miti', 'is', 'planning', 'to', 'work', 'out',
↪'a', 'revised',
 'energy', 'supply', '/', 'demand', 'outlook', 'through', 'deliberations', 'of',
↪'committee',
 'meetings', 'of', 'the', 'agency', 'of', 'natural', 'resources', 'and', 'energy', ',
↪', 'the',
 'officials', 'said', '.', 'they', 'said', 'miti', 'will', 'also', 'review', 'the',
↪'breakdown',
 'of', 'energy', 'supply', 'sources', ',', 'including', 'oil', ',', 'nuclear', ',',
↪'coal', 'and',
 'natural', 'gas', '.', 'nuclear', 'energy', 'provided', 'the', 'bulk', 'of', 'japan
↪', "'", 's',
 'electric', 'power', 'in', 'the', 'fiscal', 'year', 'ended', 'march', '31', ',',
↪'supplying',
 'an', 'estimated', '27', 'pct', 'on', 'a', 'kilowatt', '/', 'hour', 'basis', ',',
↪'followed',
 'by', 'oil', '(', '23', 'pct', ')', 'and', 'liquefied', 'natural', 'gas', '(', '21',
↪ 'pct', '),',
 'they', 'noted', '.', '<END>'],
['<START>', 'energy', '/', 'u', '.', 's', '.', 'petrochemical', 'industry', 'cheap',
↪'oil',
 'feedstocks', ',', 'the', 'weakened', 'u', '.', 's', '.', 'dollar', 'and', 'a',
↪'plant',
 'utilization', 'rate', 'approaching', '90', 'pct', 'will', 'propel', 'the',
↪'streamlined', 'u',
 '.', 's', '.', 'petrochemical', 'industry', 'to', 'record', 'profits', 'this', 'year
↪', ',',
 'with', 'growth', 'expected', 'through', 'at', 'least', '1990', ',', 'major',
↪'company',
 'executives', 'predicted', '.', 'this', 'bullish', 'outlook', 'for', 'chemical',
↪'manufacturing',
 'and', 'an', 'industrywide', 'move', 'to', 'shed', 'unrelated', 'businesses', 'has',
↪ 'prompted',
 'gaf', 'corp', '&', 'lt', ';', 'gaf', '>,', 'privately', '-', 'held', 'cain',
↪'chemical', 'inc',
 ',', 'and', 'other', 'firms', 'to', 'aggressively', 'seek', 'acquisitions', 'of',
↪'petrochemical',
 'plants', '.', 'oil', 'companies', 'such', 'as', 'ashland', 'oil', 'inc', '&', 'lt',
↪ ';', 'ash',
```

(continues on next page)

(continued from previous page)

```
 '>,', 'the', 'kentucky', '-', 'based', 'oil', 'refiner', 'and', 'marketer', ',',
↪'are', 'also',
 'shopping', 'for', 'money', '-', 'making', 'petrochemical', 'businesses', 'to', 'buy
↪', '.', '"',
 'i', 'see', 'us', 'poised', 'at', 'the', 'threshold', 'of', 'a', 'golden', 'period',
↪ ,"', 'said',
 'paul', 'oreffice', ',', 'chairman', 'of', 'giant', 'dow', 'chemical', 'co', '&',
↪'lt', ';',
 'dow', '>,', 'adding', ',', '"', 'there', "'", 's', 'no', 'major', 'plant',
↪'capacity', 'being',
 'added', 'around', 'the', 'world', 'now', '.', 'the', 'whole', 'game', 'is',
↪'bringing', 'out',
 'new', 'products', 'and', 'improving', 'the', 'old', 'ones', '."', 'analysts', 'say
↪', 'the',
 'chemical', 'industry', "'", 's', 'biggest', 'customers', ',', 'automobile',
↪'manufacturers',
 'and', 'home', 'builders', 'that', 'use', 'a', 'lot', 'of', 'paints', 'and',
↪'plastics', ',',
 'are', 'expected', 'to', 'buy', 'quantities', 'this', 'year', '.', 'u', '.', 's', '.
↪',
 'petrochemical', 'plants', 'are', 'currently', 'operating', 'at', 'about', '90',
↪'pct',
 'capacity', ',', 'reflecting', 'tighter', 'supply', 'that', 'could', 'hike',
↪'product', 'prices',
 'by', '30', 'to', '40', 'pct', 'this', 'year', ',', 'said', 'john', 'dosher', ',',
↪'managing',
 'director', 'of', 'pace', 'consultants', 'inc', 'of', 'houston', '.', 'demand', 'for
↪', 'some',
 'products', 'such', 'as', 'styrene', 'could', 'push', 'profit', 'margins', 'up', 'by
↪', 'as',
 'much', 'as', '300', 'pct', ',', 'he', 'said', '.', 'oreffice', ',', 'speaking', 'at
↪', 'a',
 'meeting', 'of', 'chemical', 'engineers', 'in', 'houston', ',', 'said', 'dow',
↪'would', 'easily',
 'top', 'the', '741', 'mln', 'dlrs', 'it', 'earned', 'last', 'year', 'and',
↪'predicted', 'it',
 'would', 'have', 'the', 'best', 'year', 'in', 'its', 'history', '.', 'in', '1985',
↪',', 'when',
 'oil', 'prices', 'were', 'still', 'above', '25', 'dlrs', 'a', 'barrel', 'and',
↪'chemical',
 'exports', 'were', 'adversely', 'affected', 'by', 'the', 'strong', 'u', '.', 's', '.
↪', 'dollar',
 ',', 'dow', 'had', 'profits', 'of', '58', 'mln', 'dlrs', '.', '"', 'i', 'believe',
↪'the',
 'entire', 'chemical', 'industry', 'is', 'headed', 'for', 'a', 'record', 'year', 'or
↪', 'close',
 'to', 'it', ',"', 'oreffice', 'said', '.', 'gaf', 'chairman', 'samuel', 'heyman',
↪'estimated',
 'that', 'the', 'u', '.', 's', '.', 'chemical', 'industry', 'would', 'report', 'a',
↪'20', 'pct',
 'gain', 'in', 'profits', 'during', '1987', '.', 'last', 'year', ',', 'the',
↪'domestic',
 'industry', 'earned', 'a', 'total', 'of', '13', 'billion', 'dlrs', ',', 'a', '54',
↪'pct', 'leap',
 'from', '1985', '.', 'the', 'turn', 'in', 'the', 'fortunes', 'of', 'the', 'once', '-
↪', 'sickly',
 'chemical', 'industry', 'has', 'been', 'brought', 'about', 'by', 'a', 'combination',
↪ 'of', 'luck',
```

(continues on next page)

(continued from previous page)

```
 'and', 'planning', ',', 'said', 'pace', "'", 's', 'john', 'dosher', '.', 'dosher',
↪'said', 'last',
 'year', "'", 's', 'fall', 'in', 'oil', 'prices', 'made', 'feedstocks', 'dramatically
↪', 'cheaper',
 'and', 'at', 'the', 'same', 'time', 'the', 'american', 'dollar', 'was', 'weakening',
↪ 'against',
 'foreign', 'currencies', '.', 'that', 'helped', 'boost', 'u', '.', 's', '.',
↪'chemical',
 'exports', '.', 'also', 'helping', 'to', 'bring', 'supply', 'and', 'demand', 'into',
↪ 'balance',
 'has', 'been', 'the', 'gradual', 'market', 'absorption', 'of', 'the', 'extra',
↪'chemical',
 'manufacturing', 'capacity', 'created', 'by', 'middle', 'eastern', 'oil', 'producers
↪', 'in',
 'the', 'early', '1980s', '.', 'finally', ',', 'virtually', 'all', 'major', 'u', '.',
↪ 's', '.',
 'chemical', 'manufacturers', 'have', 'embarked', 'on', 'an', 'extensive', 'corporate
↪',
 'restructuring', 'program', 'to', 'mothball', 'inefficient', 'plants', ',', 'trim',
↪'the',
 'payroll', 'and', 'eliminate', 'unrelated', 'businesses', '.', 'the', 'restructuring
↪', 'touched',
 'off', 'a', 'flurry', 'of', 'friendly', 'and', 'hostile', 'takeover', 'attempts', '.
↪', 'gaf', ',',
 'which', 'made', 'an', 'unsuccessful', 'attempt', 'in', '1985', 'to', 'acquire',
↪'union',
 'carbide', 'corp', '&', 'lt', ';', 'uk', '>,', 'recently', 'offered', 'three',
↪'billion', 'dlrs',
 'for', 'borg', 'warner', 'corp', '&', 'lt', ';', 'bor', '>,', 'a', 'chicago',
↪'manufacturer',
 'of', 'plastics', 'and', 'chemicals', '.', 'another', 'industry', 'powerhouse', ',',
↪ 'w', '.',
 'r', '.', 'grace', '&', 'lt', ';', 'gra', '>', 'has', 'divested', 'its', 'retailing
↪', ',',
 'restaurant', 'and', 'fertilizer', 'businesses', 'to', 'raise', 'cash', 'for',
↪'chemical',
 'acquisitions', '.', 'but', 'some', 'experts', 'worry', 'that', 'the', 'chemical',
↪'industry',
 'may', 'be', 'headed', 'for', 'trouble', 'if', 'companies', 'continue', 'turning',
↪'their',
 'back', 'on', 'the', 'manufacturing', 'of', 'staple', 'petrochemical', 'commodities
↪', ',', 'such',
 'as', 'ethylene', ',', 'in', 'favor', 'of', 'more', 'profitable', 'specialty',
↪'chemicals',
 'that', 'are', 'custom', '-', 'designed', 'for', 'a', 'small', 'group', 'of',
↪'buyers', '.', '"',
 'companies', 'like', 'dupont', '&', 'lt', ';', 'dd', '>', 'and', 'monsanto', 'co',
↪'&', 'lt', ';',
 'mtc', '>', 'spent', 'the', 'past', 'two', 'or', 'three', 'years', 'trying', 'to',
↪'get', 'out',
 'of', 'the', 'commodity', 'chemical', 'business', 'in', 'reaction', 'to', 'how',
↪'badly', 'the',
 'market', 'had', 'deteriorated', ,"', 'dosher', 'said', '.', '"', 'but', 'i',
↪'think', 'they',
 'will', 'eventually', 'kill', 'the', 'margins', 'on', 'the', 'profitable',
↪'chemicals', 'in',
 'the', 'niche', 'market', '."', 'some', 'top', 'chemical', 'executives', 'share',
↪'the',
```

(continues on next page)

(continued from previous page)

```
 'concern', '.', '"', 'the', 'challenge', 'for', 'our', 'industry', 'is', 'to', 'keep
↪', 'from',
 'getting', 'carried', 'away', 'and', 'repeating', 'past', 'mistakes', ',"', 'gaf', "
↪'", 's',
 'heyman', 'cautioned', '.', '"', 'the', 'shift', 'from', 'commodity', 'chemicals',
↪'may', 'be',
 'ill', '-', 'advised', '.', 'specialty', businesses', 'do', 'not', 'stay', 'special
↪', 'long',
 '."', 'houston', '-', 'based', 'cain', 'chemical', ',', 'created', 'this', 'month',
↪'by', 'the',
 'sterling', 'investment', 'banking', 'group', ',', 'believes', 'it', 'can',
↪'generate', '700',
 'mln', 'dlrs', 'in', 'annual', 'sales', 'by', 'bucking', 'the', 'industry', 'trend',
↪ '.',
 'chairman', 'gordon', 'cain', ',', 'who', 'previously', 'led', 'a', 'leveraged',
↪'buyout', 'of',
 'dupont', "'", 's', 'conoco', 'inc', "'", 's', 'chemical', 'business', ',', 'has',
↪'spent', '1',
 '.', '1', 'billion', 'dlrs', 'since', 'january', 'to', 'buy', 'seven',
↪'petrochemical', 'plants',
 'along', 'the', 'texas', 'gulf', 'coast', '.', 'the', 'plants', 'produce', 'only',
↪'basic',
 'commodity', 'petrochemicals', 'that', 'are', 'the', 'building', 'blocks', 'of',
↪'specialty',
 'products', '.', '"', 'this', 'kind', 'of', 'commodity', 'chemical', 'business',
↪'will', 'never',
 'be', 'a', 'glamorous', ',', 'high', '-', 'margin', 'business', ',"', 'cain', 'said
↪', ',',
 'adding', 'that', 'demand', 'is', 'expected', 'to', 'grow', 'by', 'about', 'three',
↪'pct',
 'annually', '.', 'garo', 'armen', ',', 'an', 'analyst', 'with', 'dean', 'witter',
↪'reynolds', ',',
 'said', 'chemical', 'makers', 'have', 'also', 'benefitted', 'by', 'increasing',
↪'demand', 'for',
 'plastics', 'as', 'prices', 'become', 'more', 'competitive', 'with', 'aluminum', ',
↪', 'wood',
 'and', 'steel', 'products', '.', 'armen', 'estimated', 'the', 'upturn', 'in', 'the',
↪ 'chemical',
 'business', 'could', 'last', 'as', 'long', 'as', 'four', 'or', 'five', 'years', ',',
↪ 'provided',
 'the', 'u', '.', 's', '.', 'economy', 'continues', 'its', 'modest', 'rate', 'of',
↪'growth', '.',
 '<END>'],
['<START>', 'turkey', 'calls', 'for', 'dialogue', 'to', 'solve', 'dispute', 'turkey',
↪ 'said',
 'today', 'its', 'disputes', 'with', 'greece', ',', 'including', 'rights', 'on', 'the
↪',
 'continental', 'shelf', 'in', 'the', 'aegean', 'sea', ',', 'should', 'be', 'solved',
↪ 'through',
 'negotiations', '.', 'a', 'foreign', 'ministry', 'statement', 'said', 'the', 'latest
↪', 'crisis',
 'between', 'the', 'two', 'nato', 'members', 'stemmed', 'from', 'the', 'continental',
↪ 'shelf',
 'dispute', 'and', 'an', 'agreement', 'on', 'this', 'issue', 'would', 'effect', 'the
↪', 'security',
 ',', 'economy', 'and', 'other', 'rights', 'of', 'both', 'countries', '.', '"', 'as',
↪ 'the',
```

(continues on next page)

(continued from previous page)

```
 'issue', 'is', 'basicly', 'political', ',', 'a', 'solution', 'can', 'only', 'be',
↪'found', 'by',
 'bilateral', 'negotiations', ',"', 'the', 'statement', 'said', '.', 'greece', 'has',
↪ 'repeatedly',
 'said', 'the', 'issue', 'was', 'legal', 'and', 'could', 'be', 'solved', 'at', 'the',
 'international', 'court', 'of', 'justice', '.', 'the', 'two', 'countries',
↪'approached', 'armed',
 'confrontation', 'last', 'month', 'after', 'greece', 'announced', 'it', 'planned',
↪'oil',
 'exploration', 'work', 'in', 'the', 'aegean', 'and', 'turkey', 'said', 'it', 'would
↪', 'also',
 'search', 'for', 'oil', '.', 'a', 'face', '-', 'off', 'was', 'averted', 'when',
↪'turkey',
 'confined', 'its', 'research', 'to', 'territorrial', 'waters', '.', '"', 'the',
↪'latest',
 'crises', 'created', 'an', 'historic', 'opportunity', 'to', 'solve', 'the',
↪'disputes', 'between',
 'the', 'two', 'countries', ',"', 'the', 'foreign', 'ministry', 'statement', 'said',
↪'.', 'turkey',
 "'", 's', 'ambassador', 'in', 'athens', ',', 'nazmi', 'akiman', ',', 'was', 'due',
↪'to', 'meet',
 'prime', 'minister', 'andreas', 'papandreou', 'today', 'for', 'the', 'greek', 'reply
↪', 'to', 'a',
 'message', 'sent', 'last', 'week', 'by', 'turkish', 'prime', 'minister', 'turgut',
↪'ozal', '.',
 'the', 'contents', 'of', 'the', 'message', 'were', 'not', 'disclosed', '.', '<END>
↪']]
```

7.2.3 Find `distinct_words`

Let's write a method to work out the distinct words (word types) that occur in the corpus. You can do this with `for` loops, but it's more efficient to do it with **Python list comprehensions**. In particular, this may be useful to flatten a list of lists. If you're not familiar with Python list comprehensions in general, here's more information.

You may find it useful to use Python sets to remove duplicate words.

```
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents
        Return:
            corpus_words (list of strings): list of distinct words across the corpus,␣
↪sorted (using python 'sorted' function)
            num_corpus_words (integer): number of distinct words across the corpus
    """
    corpus_words = []
    num_corpus_words = -1

    # ------------------
    # Write your implementation here.
    corpus_words = sorted(list(set([y for x in corpus for y in x])))
#     corpus_words = [y for x in corpus for y in x]
#     corpus_words = list(set(corpus_words)) # unique words
#     corpus_words = sorted(corpus_words) # sorts
    num_corpus_words = len(corpus_words)
```

(continues on next page)

(continued from previous page)

```
    # ------------------

    return corpus_words, num_corpus_words
```

{Test-driven coding practice}

A good practice is to write test code to validate your program output against expected output.

```
# ---------------------
# Run this sanity check
# Note that this not an exhaustive check for correctness.
# Very simple tokenization using the Python string split
# with space - string.split(" ").
# ---------------------

# Define toy corpus
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END_TOKEN).
 ↪split(" "), "{} All's well that ends well {}".format(START_TOKEN, END_TOKEN).split(
 ↪" ")]
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold", "All's",
 ↪"glitters", "isn't", "well", END_TOKEN])
ans_num_corpus_words = len(ans_test_corpus_words)

# Test correct number of words
assert(num_corpus_words == ans_num_corpus_words), "Incorrect number of distinct words.
 ↪ Correct: {}. Yours: {}".format(ans_num_corpus_words, num_corpus_words)

# Test correct words
assert (test_corpus_words == ans_test_corpus_words), "Incorrect corpus_words.\
 ↪nCorrect: {}\nYours:   {}".format(str(ans_test_corpus_words), str(test_corpus_
 ↪words))

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
--------------------------------------------------------------------------------
Passed All Tests!
--------------------------------------------------------------------------------
```

7.2.4 compute_co_occurrence_matrix

Write a method that constructs a co-occurrence matrix for a certain window-size n (with a default of 4), considering words n before and n after the word in the center of the window. Here, we start to use `numpy (np)` to represent vectors, matrices, and tensors. If you're not familiar with NumPy, there's a NumPy tutorial Python NumPy tutorial.

```
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (default of␣
↪4).

        Note: Each word in a document should be at the center of a window. Words near␣
↪edges will have a smaller
              number of co-occurring words.

              For example, if we take the document "<START> All that glitters is not␣
↪gold <END>" with window size of 4,
              "All" will co-occur with "<START>", "that", "glitters", "is", and "not".

        Params:
            corpus (list of list of strings): corpus of documents
            window_size (int): size of context window
        Return:
            M (a symmetric numpy matrix of shape (number of unique words in the␣
↪corpus , number of unique words in the corpus)):
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the same as␣
↪the ordering of the words given by the distinct_words function.
            word2Ind (dict): dictionary that maps word to index (i.e. row/column␣
↪number) for matrix M.
    """
    words, num_words = distinct_words(corpus)
    M = None
    word2Ind = {}

    # ------------------
    M = np.zeros((num_words, num_words))
    word2Ind = {w : i for i, w in enumerate(words)}


    for sentence in corpus:
        for cen_w_i, cen_w in enumerate(sentence): # center word
            for con_w_i in range(cen_w_i-window_size, cen_w_i+window_size+1): #␣
↪context word
                if(cen_w_i!=con_w_i and con_w_i>=0 and con_w_i<len(sentence)):

                    ce = word2Ind[cen_w]
                    co = word2Ind[sentence[con_w_i]]

                    M[ce][co] = M[ce][co] + 1

    # ------------------

    return M, word2Ind
```

```
# ----------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
```

(continues on next page)

(continued from previous page)

```
# ----------------------

# Define toy corpus and get student's co-occurrence matrix
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END_TOKEN).
 ↪split(" "), "{} All's well that ends well {}".format(START_TOKEN, END_TOKEN).split(
 ↪" ")]
M_test, word2Ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)

# Correct M and word2Ind
M_test_ans = np.array(
    [[0., 0., 0., 0., 0., 0., 1., 0., 0., 1.,],
     [0., 0., 1., 1., 0., 0., 0., 0., 0., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 1., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 0., 1., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,],
     [1., 0., 0., 0., 0., 0., 0., 1., 0., 0.,],
     [0., 0., 0., 0., 0., 1., 1., 0., 0., 0.,],
     [0., 0., 1., 0., 1., 1., 0., 0., 0., 1.,],
     [1., 0., 0., 1., 1., 0., 0., 0., 1., 0.,]]
)
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold", "All's",
 ↪"glitters", "isn't", "well", END_TOKEN])
word2Ind_ans = dict(zip(ans_test_corpus_words, range(len(ans_test_corpus_words))))

# Test correct word2Ind
assert (word2Ind_ans == word2Ind_test), "Your word2Ind is incorrect:\nCorrect: {}\
 ↪nYours: {}".format(word2Ind_ans, word2Ind_test)

# Test correct M shape
assert (M_test.shape == M_test_ans.shape), "M matrix has incorrect shape.\nCorrect: {}
 ↪\nYours: {}".format(M_test.shape, M_test_ans.shape)

# Test correct M values
for w1 in word2Ind_ans.keys():
    idx1 = word2Ind_ans[w1]
    for w2 in word2Ind_ans.keys():
        idx2 = word2Ind_ans[w2]
        student = M_test[idx1, idx2]
        correct = M_test_ans[idx1, idx2]
        if student != correct:
            print("Correct M:")
            print(M_test_ans)
            print("Your M: ")
            print(M_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}) in␣
 ↪matrix M. Yours has {} but should have {}.".format(idx1, idx2, w1, w2, student,␣
 ↪correct))

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
--------------------------------------------------------------------------------
Passed All Tests!
--------------------------------------------------------------------------------
```

7.2.5 Using SVD to `reduce_to_k_dim`

Now let's construct a method that performs dimensionality reduction on the matrix to produce k-dimensional embeddings. Use SVD to take the top k components and produce a new matrix of k-dimensional embeddings.

Note: All of numpy, scipy, and scikit-learn (`sklearn`) provide *some* implementation of SVD, but only scipy and sklearn provide an implementation of Truncated SVD, and only sklearn provides an efficient randomized algorithm for calculating large-scale Truncated SVD. So please use sklearn.decomposition.TruncatedSVD.

```
import sklearn
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words, num_
 ↪corpus_words)
        to a matrix of dimensionality (num_corpus_words, k) using the following SVD␣
 ↪function from Scikit-Learn:
            - http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.
 ↪TruncatedSVD.html

        Params:
            M (numpy matrix of shape (number of unique words in the corpus , number␣
 ↪of unique words in the corpus)): co-occurence matrix of word counts
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of corpus words, k)): matrix of␣
 ↪k-dimensioal word embeddings.
                    In terms of the SVD from math class, this actually returns U * S
    """
    n_iters = 10     # Use this parameter in your call to `TruncatedSVD`
    M_reduced = None
    print("Running Truncated SVD over %i words..." % (M.shape[0]))

        # ------------------
        # Write your implementation here.
    M_r = sklearn.decomposition.TruncatedSVD(n_components=k,
                                             algorithm='randomized',
                                             n_iter=n_iters,
                                             random_state=None,
                                             tol=0.0)
    M_reduced = M_r.fit_transform(M)
        # ------------------

    print("Done.")
    return M_reduced
```

```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness
# In fact we only check that your M_reduced has the right dimensions.
# ---------------------

# Define toy corpus and run student code
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END_TOKEN).
 ↪split(" "), "{} All's well that ends well {}".format(START_TOKEN, END_TOKEN).split(
 ↪" ")]
M_test, word2Ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)
```

(continues on next page)

(continued from previous page)

```
M_test_reduced = reduce_to_k_dim(M_test, k=2)

# Test proper dimensions
assert (M_test_reduced.shape[0] == 10), "M_reduced has {} rows; should have {}".
 ↪format(M_test_reduced.shape[0], 10)
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have {}".
 ↪format(M_test_reduced.shape[1], 2)

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
Running Truncated SVD over 10 words...
Done.
--------------------------------------------------------------------------------
Passed All Tests!
--------------------------------------------------------------------------------
```

7.2.6 Visualise the embeddings using `plot_embeddings`

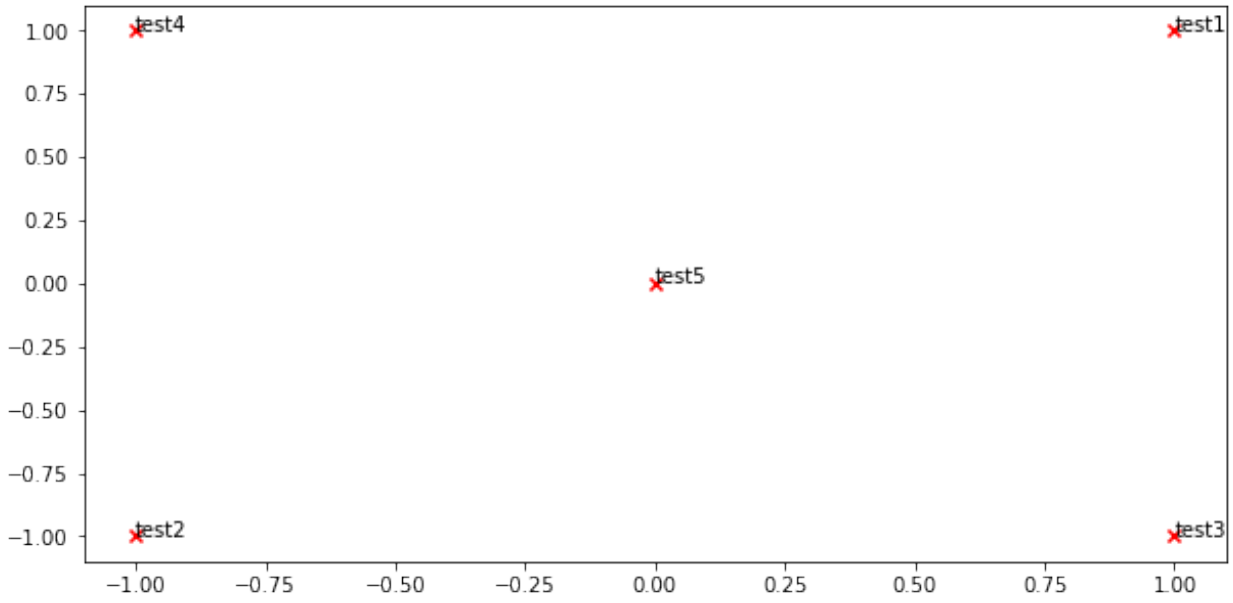
We can now write a function to plot a set of 2D vectors in 2D space. For graphs, we will use Matplotlib (`plt`).

For this example, you may find it useful to adapt this code. In the future, a good way to make a plot is to look at the Matplotlib gallery, find a plot that looks somewhat like what you want, and adapt the code they give.

```
def plot_embeddings(M_reduced, word2Ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list "words
 ↪".
        NOTE: do not plot all the words listed in M_reduced / word2Ind.
        Include a label next to each point.

        Params:
            M_reduced (numpy matrix of shape (number of unique words in the corpus ,␣
 ↪2)): matrix of 2-dimensioal word embeddings
            word2Ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visualize
    """

    # ------------------
    import matplotlib.pyplot as plt
    x_coords = M_reduced[:,0]
    y_coords = M_reduced[:,1]

    for word in words:
        i = word2Ind[word]
        x = x_coords[i]
        y = y_coords[i]
        plt.scatter(x, y, color='r', marker='x')
        plt.annotate(word, (x, y))
    plt.show()

    # ------------------
```

```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the "test solution plot" depicted below.
# ---------------------

print ("-" * 80)
print ("Outputted Plot:")

M_reduced_plot_test = np.array([[1, 1], [-1, -1], [1, -1], [-1, 1], [0, 0]])
word2Ind_plot_test = {'test1': 0, 'test2': 1, 'test3': 2, 'test4': 3, 'test5': 4}
words = ['test1', 'test2', 'test3', 'test4', 'test5']
plot_embeddings(M_reduced_plot_test, word2Ind_plot_test, words)

print ("-" * 80)
```

```
--------------------------------------------------------------------------------
Outputted Plot:
```




```
--------------------------------------------------------------------------------
```

7.3 GloVe: Global Vectors for Word Representation

As discussed in the lecture, more recently prediction-based word vectors have demonstrated better performance, such as word2vec and GloVe (which also utilizes the benefit of counts). Here, we will explore the embeddings produced by GloVe. Please revisit the lecture slides for more details on the word2vec and GloVe algorithms. If you're feeling adventurous, challenge yourself and try reading GloVe's original paper.

Run the following cells to load the GloVe vectors into memory.

Note: If this is your first time to run these cells, i.e. download the embedding model, it will take about 15 minutes to run. If you've run these cells before, rerunning them will load the model without redownloading it, which will take about

1 to 2 minutes.

Important: The code in this notebook is updated to work with Gensim v4.0.

```
# All Import Statements Defined Here
# Note: Do not add to this list.
# ----------------

import sys
assert sys.version_info[0]==3
assert sys.version_info[1] >= 5

from gensim.models import KeyedVectors
from gensim.test.utils import datapath
import pprint
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 5]
import nltk
nltk.download('reuters')
from nltk.corpus import reuters
import numpy as np
import random
import scipy as sp
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import PCA

START_TOKEN = '<START>'
END_TOKEN = '<END>'

np.random.seed(0)
random.seed(0)
# ----------------
```

```
C:\ProgramData\Anaconda3\envs\cits4012\lib\site-packages\gensim\similarities\__init__.
 ↪py:15: UserWarning: The gensim.similarities.levenshtein submodule is disabled,␣
 ↪because the optional Levenshtein package <https://pypi.org/project/python-
 ↪Levenshtein/> is unavailable. Install Levenhstein (e.g. `pip install python-
 ↪Levenshtein`) to suppress this warning.
  warnings.warn(msg)
[nltk_data] Downloading package reuters to
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

```
def load_embedding_model():
    """ Load GloVe Vectors
        Return:
            wv_from_bin: All 400000 embeddings, each lengh 200
    """
    import gensim.downloader as api
    wv_pretrained = api.load("glove-wiki-gigaword-200")
    print("Loaded vocab size %i" % len(wv_pretrained))
    print("The loaded object is of type %s" % str(type(wv_pretrained)))
    return wv_pretrained
```

```
# ----------------------------------
# Run Cell to Load Word Vectors
# Note: This will take several minutes
# (8 mins in my case )
# ----------------------------------
wv_pretrained = load_embedding_model()
```

```
Loaded vocab size 400000
The loaded object is of type <class 'gensim.models.keyedvectors.KeyedVectors'>
```

Important: Gensim 4.0 has made some significant changes to its syntax and object oragnisation. For more information see Magrating from Gensim 3.x to 4

The `KeyedVectors` object has a dictionary property (`.key_to_index`) that returns a disctionary mapping of words to indices.

```
{key:value for key, value in wv_pretrained.key_to_index.items() if value<10}
```

```
{'the': 0,
 ',': 1,
 '.': 2,
 'of': 3,
 'to': 4,
 'and': 5,
 'in': 6,
 'a': 7,
 '"': 8,
 "'s": 9}
```

```
wv_pretrained.get_vector('engineering')
```

```
array([-5.1938e-02,  1.7635e-01,  1.8786e-01, -4.7958e-01,  1.0851e+00,
       -2.8156e-01, -2.8066e-02,  9.1940e-02, -7.6533e-02,  3.4757e-02,
       -5.2543e-02,  3.4220e-01, -3.3942e-01,  1.2675e-01,  3.4639e-02,
       -2.7710e-01, -1.1907e-01, -2.2184e-01,  7.1293e-01, -2.9338e-01,
       -4.2741e-01,  2.1376e+00,  1.0362e+00, -1.7944e-01,  6.5682e-01,
        5.7012e-01, -1.8262e-01,  4.5973e-01,  7.9056e-01, -8.7835e-01,
        7.9399e-01,  3.0741e-02, -1.3160e-01, -2.8041e-01,  1.3836e-01,
        4.1736e-01,  8.1038e-05, -5.7410e-01, -5.9988e-02,  1.2367e-01,
       -6.9972e-01, -3.6496e-01, -2.8456e-01, -4.4996e-01,  3.0715e-01,
       -6.0679e-01,  2.5705e-01, -3.8619e-01, -4.6077e-02, -1.7457e-01,
        3.8706e-01,  4.0831e-01, -1.7162e-01, -2.3532e-02, -3.5146e-01,
       -3.1349e-01,  2.8886e-02,  5.0182e-01,  3.4199e-01, -3.4627e-01,
        7.0743e-01,  1.5336e-01,  5.5904e-01,  4.0061e-01, -2.4110e-01,
       -3.8127e-01, -3.7064e-01,  1.1633e+00, -2.7004e-01,  3.4658e-01,
        2.3165e-01, -8.0207e-01,  3.9961e-01,  1.3949e-01, -2.1591e-01,
       -1.7998e-01,  7.4177e-01,  4.7986e-02, -2.5916e-01, -3.4934e-01,
        6.1844e-01,  1.9083e-03, -4.6262e-01,  7.7001e-01,  3.2828e-01,
       -3.2410e-01, -4.4820e-01,  4.7219e-01,  1.6762e+00,  3.1895e-01,
        5.2878e-01, -5.3253e-01,  1.3662e-01,  3.6778e-01, -7.2084e-01,
       -6.4639e-01,  2.8320e-01, -2.6098e-01,  7.2896e-01,  5.0380e-01,
        1.1000e-01, -1.0159e-01, -4.8650e-01,  1.0542e+00, -6.7540e-02,
        4.6954e-01, -6.2859e-02,  8.7161e-01,  2.6370e-01,  4.8938e-01,
       -1.0615e-02,  1.9904e-01,  9.3244e-02,  6.9493e-01, -2.6162e-01,
```

(continues on next page)

(continued from previous page)

```
        3.7821e-01,  8.9474e-02,  4.8153e-01,  4.5557e-02,  8.0432e-02,
        1.0388e-01, -9.1506e-01, -4.7667e-01,  1.7653e-02, -2.0044e-01,
        1.1785e-01, -1.6728e-01, -4.9748e-02, -9.6106e-02, -3.5579e-02,
       -1.4029e+00,  1.4633e-01, -3.3947e-02, -1.0846e+00, -7.8798e-01,
        8.8403e-02, -3.0667e-01,  8.8800e-01, -6.6901e-02,  7.8241e-01,
        6.1223e-02,  8.2684e-03, -2.9807e-01, -1.0423e-01,  4.9161e-01,
       -2.1850e-01, -1.2854e-01, -2.8894e-02,  1.1280e+00, -3.7454e-01,
        2.6395e-01, -7.8308e-01,  2.5786e-01, -8.8650e-01,  1.3409e-01,
       -3.8980e-01,  2.5154e-01, -6.2761e-02, -1.7067e-02, -7.6850e-01,
        7.6643e-01,  2.0544e-01, -2.2464e-01,  1.9980e-01, -3.8998e-01,
        7.6401e-01, -5.1074e-01, -3.2532e-01,  1.7820e-01,  9.8373e-01,
        9.9213e-01,  2.4715e-01,  1.3402e-01,  2.7492e-01, -5.4361e-02,
       -1.8243e-01,  2.1020e-02, -2.6531e-01, -1.8704e-01,  4.5575e-01,
        7.4159e-01, -1.2683e-02,  2.7622e-01,  3.3264e-01,  4.9330e-01,
       -1.2482e-01, -5.5854e-01,  4.4076e-01,  6.5997e-01,  2.7176e-01,
        2.2405e-01,  6.3891e-01,  4.1986e-01,  3.5628e-01,  2.1770e-01,
        2.9551e-01,  3.1118e-01, -7.8597e-02,  5.5859e-01, -1.4067e-04],
      dtype=float32)
```

7.3.1 Note: If you are receiving reset by peer error, rerun the cell to restart the download.

7.3.2 Reducing dimensionality of Word Embeddings

Let's directly compare the GloVe embeddings to those of the co-occurrence matrix. In order to avoid running out of memory, we will work with a sample of 10000 GloVe vectors instead. Run the following cells to:

1. Put 10000 Glove vectors into a matrix M
2. Run reduce_to_k_dim (your Truncated SVD function) to reduce the vectors from 200-dimensional to 2-dimensional.

To import functions from another Jupyter notebook, we will use the `import_ipynb package`. You will need to install this into your environment. " pip install import_ipynb "

```
def get_matrix_of_vectors(wv_pretrained, required_words=['barrels', 'bpd', 'ecuador',
↪'energy', 'industry', 'kuwait', 'oil', 'output', 'petroleum', 'venezuela']):
    """ Put the GloVe vectors into a matrix M.
        Param:
            wv_pretrained: KeyedVectors object; the 400000 GloVe vectors loaded from␣
↪file
        Return:
            M: numpy matrix shape (num words, 200) containing the vectors
            word2Ind: dictionary mapping each word to its row number in M
    """
    import random
    words = list(wv_pretrained.key_to_index.keys())
    print("Shuffling words ...")
    random.seed(224)
    random.shuffle(words)
    words = words[:10000]
    print("Putting %i words into word2Ind and matrix M..." % len(words))
    word2Ind = {}
    M = []
    curInd = 0
```

(continues on next page)

(continued from previous page)

```
    for w in words:
        try:
            M.append(wv_pretrained.get_vector(w, norm=True))
            word2Ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    for w in required_words:
        if w in words:
            continue
        try:
            M.append(wv_pretrained.get_vector(w, norm=True))
            word2Ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    M = np.stack(M)
    print("Done.")
    return M, word2Ind
```

```
import import_ipynb
from svd import reduce_to_k_dim

# -----------------------------------------------------------------
# Run Cell to Reduce 200-Dimensional Word Embeddings to k Dimensions
# Note: This should be quick to run
# -----------------------------------------------------------------
M, word2Ind = get_matrix_of_vectors(wv_pretrained)
M_reduced = reduce_to_k_dim(M, k=2)

# Rescale (normalize) the rows to make them each of unit-length
M_lengths = np.linalg.norm(M_reduced, axis=1)
M_reduced_normalized = M_reduced / M_lengths[:, np.newaxis] # broadcasting
```

```
importing Jupyter notebook from svd.ipynb
```

```
[nltk_data] Downloading package reuters to
[nltk_data]     C:\Users\wei\AppData\Roaming\nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

```
[['<START>', 'japan', 'to', 'revise', 'long', '-', 'term', 'energy', 'demand',
 ↪'downwards', 'the',
  'ministry', 'of', 'international', 'trade', 'and', 'industry', '(', 'miti', ')',
 ↪'will', 'revise',
  'its', 'long', '-', 'term', 'energy', 'supply', '/', 'demand', 'outlook', 'by',
 ↪'august', 'to',
  'meet', 'a', 'forecast', 'downtrend', 'in', 'japanese', 'energy', 'demand', ',',
 ↪'ministry',
  'officials', 'said', '.', 'miti', 'is', 'expected', 'to', 'lower', 'the',
 ↪'projection', 'for',
  'primary', 'energy', 'supplies', 'in', 'the', 'year', '2000', 'to', '550', 'mln',
 ↪'kilolitres',
  '(', 'kl', ')', 'from', '600', 'mln', ',', 'they', 'said', '.', 'the', 'decision',
 ↪'follows',
  'the', 'emergence', 'of', 'structural', 'changes', 'in', 'japanese', 'industry',
 ↪'following',
```

(continues on next page)

(continued from previous page)

```
 'the', 'rise', 'in', 'the', 'value', 'of', 'the', 'yen', 'and', 'a', 'decline', 'in
↪', 'domestic',
 'electric', 'power', 'demand', '.', 'miti', 'is', 'planning', 'to', 'work', 'out',
↪'a', 'revised',
 'energy', 'supply', '/', 'demand', 'outlook', 'through', 'deliberations', 'of',
↪'committee',
 'meetings', 'of', 'the', 'agency', 'of', 'natural', 'resources', 'and', 'energy', ',
↪', 'the',
 'officials', 'said', '.', 'they', 'said', 'miti', 'will', 'also', 'review', 'the',
↪'breakdown',
 'of', 'energy', 'supply', 'sources', ',', 'including', 'oil', ',', 'nuclear', ',',
↪'coal', 'and',
 'natural', 'gas', '.', 'nuclear', 'energy', 'provided', 'the', 'bulk', 'of', 'japan
↪', "'", 's',
 'electric', 'power', 'in', 'the', 'fiscal', 'year', 'ended', 'march', '31', ',',
↪'supplying',
 'an', 'estimated', '27', 'pct', 'on', 'a', 'kilowatt', '/', 'hour', 'basis', ',',
↪'followed',
 'by', 'oil', '(', '23', 'pct', ')', 'and', 'liquefied', 'natural', 'gas', '(', '21',
↪ 'pct', '),',
 'they', 'noted', '.', '<END>'],
['<START>', 'energy', '/', 'u', '.', 's', '.', 'petrochemical', 'industry', 'cheap',
↪'oil',
 'feedstocks', ',', 'the', 'weakened', 'u', '.', 's', '.', 'dollar', 'and', 'a',
↪'plant',
 'utilization', 'rate', 'approaching', '90', 'pct', 'will', 'propel', 'the',
↪'streamlined', 'u',
 '.', 's', '.', 'petrochemical', 'industry', 'to', 'record', 'profits', 'this', 'year
↪', ',',
 'with', 'growth', 'expected', 'through', 'at', 'least', '1990', ',', 'major',
↪'company',
 'executives', 'predicted', '.', 'this', 'bullish', 'outlook', 'for', 'chemical',
↪'manufacturing',
 'and', 'an', 'industrywide', 'move', 'to', 'shed', 'unrelated', 'businesses', 'has',
↪ 'prompted',
 'gaf', 'corp', '&', 'lt', ';', 'gaf', '>,', 'privately', '-', 'held', 'cain',
↪'chemical', 'inc',
 ',', 'and', 'other', 'firms', 'to', 'aggressively', 'seek', 'acquisitions', 'of',
↪'petrochemical',
 'plants', '.', 'oil', 'companies', 'such', 'as', 'ashland', 'oil', 'inc', '&', 'lt',
↪ ';', 'ash',
 '>,', 'the', 'kentucky', '-', 'based', 'oil', 'refiner', 'and', 'marketer', ',',
↪'are', 'also',
 'shopping', 'for', 'money', '-', 'making', 'petrochemical', 'businesses', 'to', 'buy
↪', '.', '"',
 'i', 'see', 'us', 'poised', 'at', 'the', 'threshold', 'of', 'a', 'golden', 'period',
↪ ',"', 'said',
 'paul', 'oreffice', ',', 'chairman', 'of', 'giant', 'dow', 'chemical', 'co', '&',
↪'lt', ';',
 'dow', '>,', 'adding', ',', '"', 'there', "'", 's', 'no', 'major', 'plant',
↪'capacity', 'being',
 'added', 'around', 'the', 'world', 'now', '.', 'the', 'whole', 'game', 'is',
↪'bringing', 'out',
 'new', 'products', 'and', 'improving', 'the', 'old', 'ones', '."', 'analysts', 'say
↪', 'the',
 'chemical', 'industry', "'", 's', 'biggest', 'customers', ',', 'automobile',
↪'manufacturers',
```

(continues on next page)

(continued from previous page)

```
 'and', 'home', 'builders', 'that', 'use', 'a', 'lot', 'of', 'paints', 'and',
↪'plastics', ',',
 'are', 'expected', 'to', 'buy', 'quantities', 'this', 'year', '.', 'u', '.', 's', '.
↪',
 'petrochemical', 'plants', 'are', 'currently', 'operating', 'at', 'about', '90',
↪'pct',
 'capacity', ',', 'reflecting', 'tighter', 'supply', 'that', 'could', 'hike',
↪'product', 'prices',
 'by', '30', 'to', '40', 'pct', 'this', 'year', ',', 'said', 'john', 'dosher', ',',
↪'managing',
 'director', 'of', 'pace', 'consultants', 'inc', 'of', 'houston', '.', 'demand', 'for
↪', 'some',
 'products', 'such', 'as', 'styrene', 'could', 'push', 'profit', 'margins', 'up', 'by
↪', 'as',
 'much', 'as', '300', 'pct', ',', 'he', 'said', '.', 'oreffice', ',', 'speaking', 'at
↪', 'a',
 'meeting', 'of', 'chemical', 'engineers', 'in', 'houston', ',', 'said', 'dow',
↪'would', 'easily',
 'top', 'the', '741', 'mln', 'dlrs', 'it', 'earned', 'last', 'year', 'and',
↪'predicted', 'it',
 'would', 'have', 'the', 'best', 'year', 'in', 'its', 'history', '.', 'in', '1985',
↪',', 'when',
 'oil', 'prices', 'were', 'still', 'above', '25', 'dlrs', 'a', 'barrel', 'and',
↪'chemical',
 'exports', 'were', 'adversely', 'affected', 'by', 'the', 'strong', 'u', '.', 's', '.
↪', 'dollar',
 ',', 'dow', 'had', 'profits', 'of', '58', 'mln', 'dlrs', '.', '"', 'i', 'believe',
↪'the',
 'entire', 'chemical', 'industry', 'is', 'headed', 'for', 'a', 'record', 'year', 'or
↪', 'close',
 'to', 'it', ',"', 'oreffice', 'said', '.', 'gaf', 'chairman', 'samuel', 'heyman',
↪'estimated',
 'that', 'the', 'u', '.', 's', '.', 'chemical', 'industry', 'would', 'report', 'a',
↪'20', 'pct',
 'gain', 'in', 'profits', 'during', '1987', '.', 'last', 'year', ',', 'the',
↪'domestic',
 'industry', 'earned', 'a', 'total', 'of', '13', 'billion', 'dlrs', ',', 'a', '54',
↪'pct', 'leap',
 'from', '1985', '.', 'the', 'turn', 'in', 'the', 'fortunes', 'of', 'the', 'once', '-
↪', 'sickly',
 'chemical', 'industry', 'has', 'been', 'brought', 'about', 'by', 'a', 'combination',
↪ 'of', 'luck',
 'and', 'planning', ',', 'said', 'pace', "'", 's', 'john', 'dosher', '.', 'dosher',
↪'said', 'last',
 'year', "'", 's', 'fall', 'in', 'oil', 'prices', 'made', 'feedstocks', 'dramatically
↪', 'cheaper',
 'and', 'at', 'the', 'same', 'time', 'the', 'american', 'dollar', 'was', 'weakening',
↪ 'against',
 'foreign', 'currencies', '.', 'that', 'helped', 'boost', 'u', '.', 's', '.',
↪'chemical',
 'exports', '.', 'also', 'helping', 'to', 'bring', 'supply', 'and', 'demand', 'into',
↪ 'balance',
 'has', 'been', 'the', 'gradual', 'market', 'absorption', 'of', 'the', 'extra',
↪'chemical',
 'manufacturing', 'capacity', 'created', 'by', 'middle', 'eastern', 'oil', 'producers
↪', 'in',
 'the', 'early', '1980s', '.', 'finally', ',', 'virtually', 'all', 'major', 'u', '.',
↪ 's', '.',
```

(continues on next page)

(continued from previous page)

```
 'chemical', 'manufacturers', 'have', 'embarked', 'on', 'an', 'extensive', 'corporate
↪',
 'restructuring', 'program', 'to', 'mothball', 'inefficient', 'plants', ',', 'trim',
↪'the',
 'payroll', 'and', 'eliminate', 'unrelated', 'businesses', '.', 'the', 'restructuring
↪', 'touched',
 'off', 'a', 'flurry', 'of', 'friendly', 'and', 'hostile', 'takeover', 'attempts', '.
↪', 'gaf', ',',
 'which', 'made', 'an', 'unsuccessful', 'attempt', 'in', '1985', 'to', 'acquire',
↪'union',
 'carbide', 'corp', '&', 'lt', ';', 'uk', '>,', 'recently', 'offered', 'three',
↪'billion', 'dlrs',
 'for', 'borg', 'warner', 'corp', '&', 'lt', ';', 'bor', '>,', 'a', 'chicago',
↪'manufacturer',
 'of', 'plastics', 'and', 'chemicals', '.', 'another', 'industry', 'powerhouse', ',',
↪ 'w', '.',
 'r', '.', 'grace', '&', 'lt', ';', 'gra', '>', 'has', 'divested', 'its', 'retailing
↪', ',',
 'restaurant', 'and', 'fertilizer', 'businesses', 'to', 'raise', 'cash', 'for',
↪'chemical',
 'acquisitions', '.', 'but', 'some', 'experts', 'worry', 'that', 'the', 'chemical',
↪'industry',
 'may', 'be', 'headed', 'for', 'trouble', 'if', 'companies', 'continue', 'turning',
↪'their',
 'back', 'on', 'the', 'manufacturing', 'of', 'staple', 'petrochemical', 'commodities
↪', ',', 'such',
 'as', 'ethylene', ',', 'in', 'favor', 'of', 'more', 'profitable', 'specialty',
↪'chemicals',
 'that', 'are', 'custom', '-', 'designed', 'for', 'a', 'small', 'group', 'of',
↪'buyers', '.', '"',
 'companies', 'like', 'dupont', '&', 'lt', ';', 'dd', '>', 'and', 'monsanto', 'co',
↪'&', 'lt', ';',
 'mtc', '>', 'spent', 'the', 'past', 'two', 'or', 'three', 'years', 'trying', 'to',
↪'get', 'out',
 'of', 'the', 'commodity', 'chemical', 'business', 'in', 'reaction', 'to', 'how',
↪'badly', 'the',
 'market', 'had', 'deteriorated', ',"', 'dosher', 'said', '.', '"', 'but', 'i',
↪'think', 'they',
 'will', 'eventually', 'kill', 'the', 'margins', 'on', 'the', 'profitable',
↪'chemicals', 'in',
 'the', 'niche', 'market', '."', 'some', 'top', 'chemical', 'executives', 'share',
↪'the',
 'concern', '.', '"', 'the', 'challenge', 'for', 'our', 'industry', 'is', 'to', 'keep
↪', 'from',
 'getting', 'carried', 'away', 'and', 'repeating', 'past', 'mistakes', ',"', 'gaf', "
↪'", 's',
 'heyman', 'cautioned', '.', '"', 'the', 'shift', 'from', 'commodity', 'chemicals',
↪'may', 'be',
 'ill', '-', 'advised', '.', 'specialty', 'businesses', 'do', 'not', 'stay', 'special
↪', 'long',
 '."', 'houston', '-', 'based', 'cain', 'chemical', ',', 'created', 'this', 'month',
↪'by', 'the',
 'sterling', 'investment', 'banking', 'group', ',', 'believes', 'it', 'can',
↪'generate', '700',
 'mln', 'dlrs', 'in', 'annual', 'sales', 'by', 'bucking', 'the', 'industry', 'trend',
↪ '.',
 'chairman', 'gordon', 'cain', ',', 'who', 'previously', 'led', 'a', 'leveraged',
↪'buyout', 'of',
```

(continues on next page)

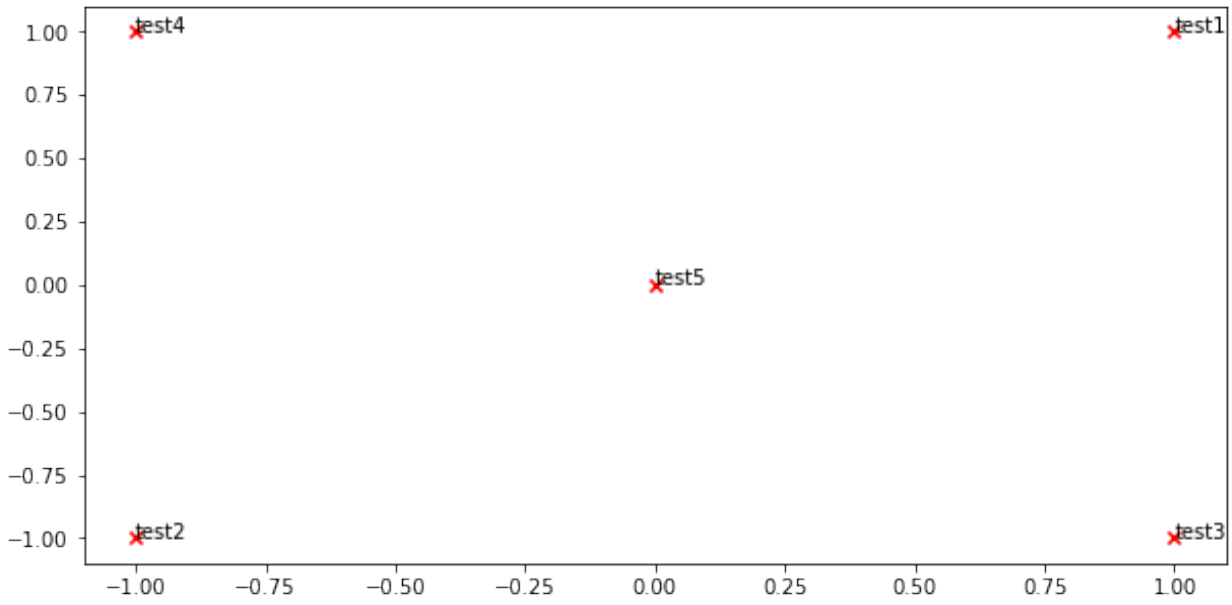
(continued from previous page)

```
 'dupont', "'", 's', 'conoco', 'inc', "'", 's', 'chemical', 'business', ',', 'has',
↪'spent', '1',
 '.', '1', 'billion', 'dlrs', 'since', 'january', 'to', 'buy', 'seven',
↪'petrochemical', 'plants',
 'along', 'the', 'texas', 'gulf', 'coast', '.', 'the', 'plants', 'produce', 'only',
↪'basic',
 'commodity', 'petrochemicals', 'that', 'are', 'the', 'building', 'blocks', 'of',
↪'specialty',
 'products', '.', '"', 'this', 'kind', 'of', 'commodity', 'chemical', 'business',
↪'will', 'never',
 'be', 'a', 'glamorous', ',', 'high', '-', 'margin', 'business', ,"', 'cain', 'said
↪', ',',
 'adding', 'that', 'demand', 'is', 'expected', 'to', 'grow', 'by', 'about', 'three',
↪'pct',
 'annually', '.', 'garo', 'armen', ',', 'an', 'analyst', 'with', 'dean', 'witter',
↪'reynolds', ',',
 'said', 'chemical', 'makers', 'have', 'also', 'benefitted', 'by', 'increasing',
↪'demand', 'for',
 'plastics', 'as', 'prices', 'become', 'more', 'competitive', 'with', 'aluminum', ',
↪', 'wood',
 'and', 'steel', 'products', '.', 'armen', 'estimated', 'the', 'upturn', 'in', 'the',
↪ 'chemical',
 'business', 'could', 'last', 'as', 'long', 'as', 'four', 'or', 'five', 'years', ',',
↪ 'provided',
 'the', 'u', '.', 's', '.', 'economy', 'continues', 'its', 'modest', 'rate', 'of',
↪'growth', '.',
 '<END>'],
['<START>', 'turkey', 'calls', 'for', 'dialogue', 'to', 'solve', 'dispute', 'turkey',
↪ 'said',
 'today', 'its', 'disputes', 'with', 'greece', ',', 'including', 'rights', 'on', 'the
↪',
 'continental', 'shelf', 'in', 'the', 'aegean', 'sea', ',', 'should', 'be', 'solved',
↪ 'through',
 'negotiations', '.', 'a', 'foreign', 'ministry', 'statement', 'said', 'the', 'latest
↪', 'crisis',
 'between', 'the', 'two', 'nato', 'members', 'stemmed', 'from', 'the', 'continental',
↪ 'shelf',
 'dispute', 'and', 'an', 'agreement', 'on', 'this', 'issue', 'would', 'effect', 'the
↪', 'security',
 ',', 'economy', 'and', 'other', 'rights', 'of', 'both', 'countries', '.', '"', 'as',
↪ 'the',
 'issue', 'is', 'basicly', 'political', ',', 'a', 'solution', 'can', 'only', 'be',
↪'found', 'by',
 'bilateral', 'negotiations', ,"', 'the', 'statement', 'said', '.', 'greece', 'has',
↪ 'repeatedly',
 'said', 'the', 'issue', 'was', 'legal', 'and', 'could', 'be', 'solved', 'at', 'the',
 'international', 'court', 'of', 'justice', '.', 'the', 'two', 'countries',
↪'approached', 'armed',
 'confrontation', 'last', 'month', 'after', 'greece', 'announced', 'it', 'planned',
↪'oil',
 'exploration', 'work', 'in', 'the', 'aegean', 'and', 'turkey', 'said', 'it', 'would
↪', 'also',
 'search', 'for', 'oil', '.', 'a', 'face', '-', 'off', 'was', 'averted', 'when',
↪'turkey',
 'confined', 'its', 'research', 'to', 'territorrial', 'waters', '.', '"', 'the',
↪'latest',
 'crises', 'created', 'an', 'historic', 'opportunity', 'to', 'solve', 'the',
↪'disputes', 'between',
```

(continues on next page)

(continued from previous page)

```
  'the', 'two', 'countries', ',"', 'the', 'foreign', 'ministry', 'statement', 'said',
 ↪'.', 'turkey',
  "'", 's', 'ambassador', 'in', 'athens', ',', 'nazmi', 'akiman', ',', 'was', 'due',
 ↪'to', 'meet',
  'prime', 'minister', 'andreas', 'papandreou', 'today', 'for', 'the', 'greek', 'reply
 ↪', 'to', 'a',
  'message', 'sent', 'last', 'week', 'by', 'turkish', 'prime', 'minister', 'turgut',
 ↪'ozal', '.',
  'the', 'contents', 'of', 'the', 'message', 'were', 'not', 'disclosed', '.', '<END>
 ↪']]
--------------------------------------------------------------------------------
Passed All Tests!
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Passed All Tests!
--------------------------------------------------------------------------------
Running Truncated SVD over 10 words...
Done.
--------------------------------------------------------------------------------
Passed All Tests!
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Outputted Plot:
```




```
--------------------------------------------------------------------------------
Shuffling words ...
Putting 10000 words into word2Ind and matrix M...
Done.
Running Truncated SVD over 10010 words...
Done.
```

Note: If you are receiving out of memory issues on your local machine, try closing other applications to free more memory on your device. You may want to try restarting your machine so that you can free up extra memory. Then

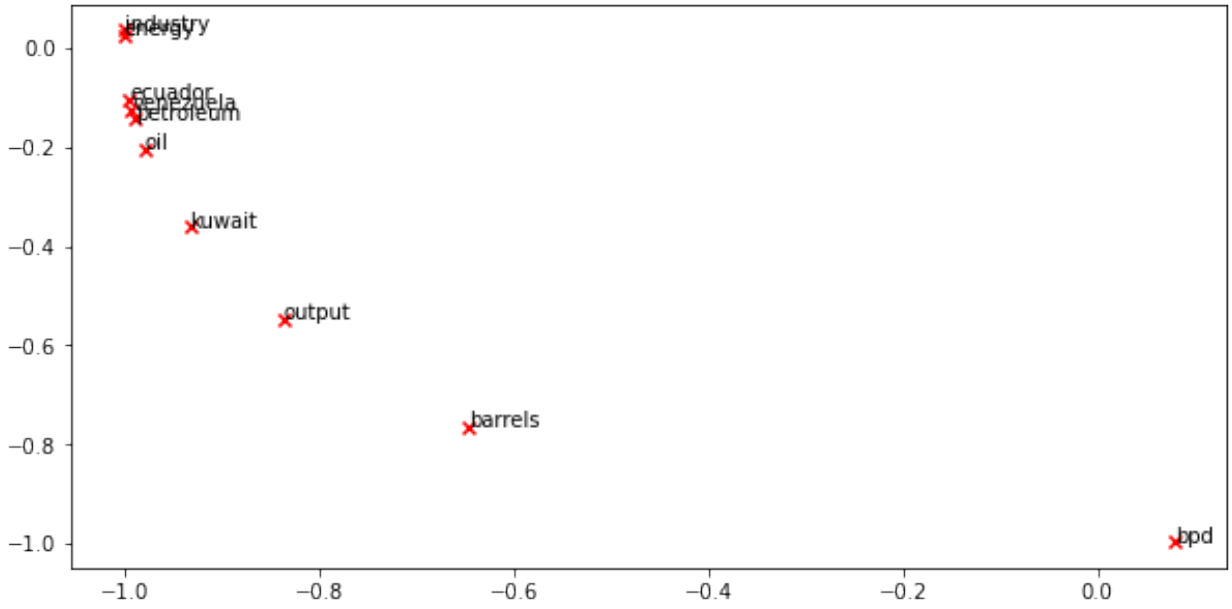
immediately run the jupyter notebook and see if you can load the word vectors properly.

7.3.3 GloVe Plot Analysis

Run the cell below to plot the 2D GloVe embeddings for `['barrels', 'bpd', 'ecuador', 'energy', 'industry', 'kuwait', 'oil', 'output', 'petroleum', 'venezuela']`.

What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have? How is the plot different from the one generated earlier from the co-occurrence matrix? What is a possible reason for causing the difference?

```
from svd import plot_embeddings

words = ['barrels', 'bpd', 'ecuador', 'energy', 'industry', 'kuwait', 'oil', 'output',
 ↪ 'petroleum', 'venezuela']
plot_embeddings(M_reduced_normalized, word2Ind, words)
```



7.4 Word2Vec

7.4.1 Using Pretrained Word2Vec Vectors

Gensim has functions to support the downloading of pretrained Word2Vec vectors. Let's use another visualisation tool (`sklearn.manifold.TSNE`) to see how the word embeddings can help us identify clusters of words with similar meaning.

```
from sklearn.manifold import TSNE
import numpy as np
import matplotlib.pyplot as plt
```

```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

Analogy Test: A is to B as C is to D

```
king - man + woman = ?
```

Google has released a analogy test set, you can use it to evaluate the performance of word embeddings.

Your Turn

Try some other set of words and see how well the pretrained embeddings perform. Consider writing an evaluation function using the analogy test set. If the first word returned by the analogy test matches what's in the test set, it's a HIT. As we only consider the most similar for comparison, this evaluation is called HIT@1.

```
wv.most_similar(positive=["king", "woman"], negative=["man"])
```

```
[('queen', 0.7118193507194519),
 ('monarch', 0.6189674735069275),
 ('princess', 0.5902431011199951),
 ('crown_prince', 0.5499460697174072),
 ('prince', 0.5377322435379028),
 ('kings', 0.5236844420433044),
 ('Queen_Consort', 0.5235945582389832),
 ('queens', 0.5181134939193726),
 ('sultan', 0.5098593235015869),
 ('monarchy', 0.5087411403656006)]
```

Plot using TSNE

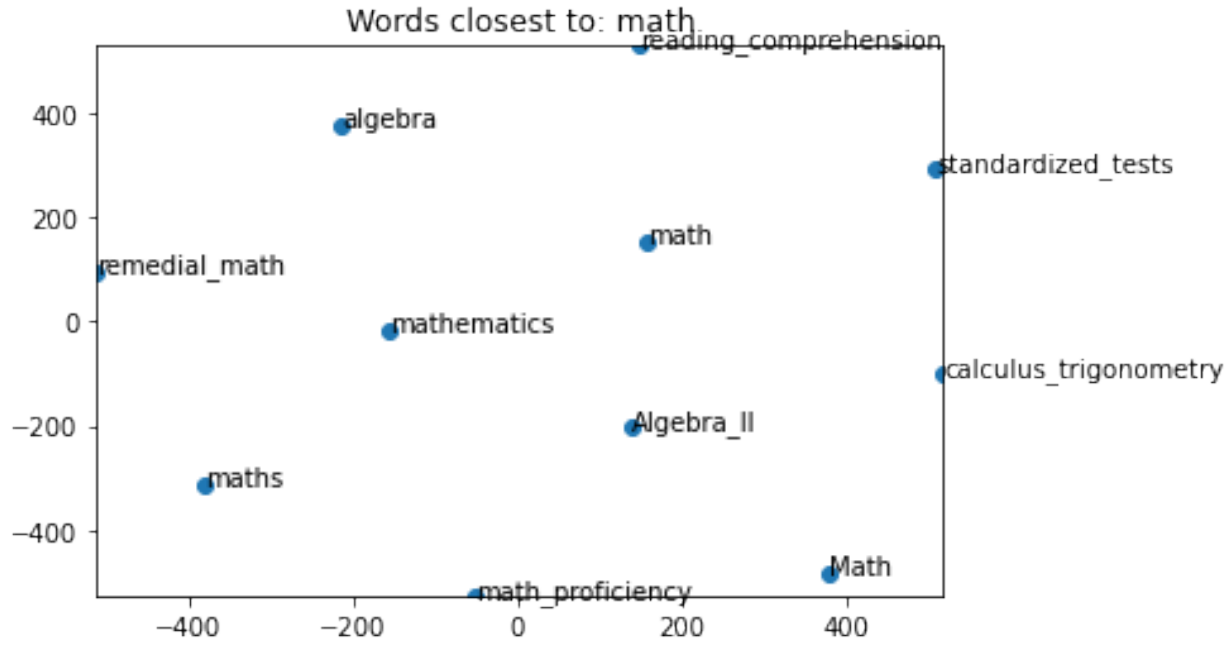
```
def display_closestwords_tsnescatterplot(model, word):

    arr = np.empty((0,300), dtype='f')
    word_labels = [word]

    # get close words
    close_words = model.similar_by_word(word)

    # add the vector for each of the closest words to the array
    arr = np.append(arr, np.array([model[word]]), axis=0)
    for wrd_score in close_words:
        wrd_vector = model[wrd_score[0]]
        word_labels.append(wrd_score[0])
        arr = np.append(arr, np.array([wrd_vector]), axis=0)

    # find tsne coords for 2 dimensions
    tsne = TSNE(n_components=2, random_state=0)
    np.set_printoptions(suppress=True)
    Y = tsne.fit_transform(arr)

    x_coords = Y[:, 0]
```

(continues on next page)

(continued from previous page)

```
    y_coords = Y[:, 1]
    # display scatter plot
    plt.scatter(x_coords, y_coords)

    for label, x, y in zip(word_labels, x_coords, y_coords):
        plt.annotate(label, xy=(x, y), xytext=(0, 0), textcoords='offset points')
    plt.xlim(x_coords.min()+0.00005, x_coords.max()+0.00005)
    plt.ylim(y_coords.min()+0.00005, y_coords.max()+0.00005)
    plt.title(f'Words closest to: {word}')
    plt.show()
```

```
display_closestwords_tsnescatterplot(wv, "math")
```



Interactive Visualisation using bokeh

```
from bokeh.plotting import figure, show
from bokeh.io import push_notebook, output_notebook
from bokeh.models import ColumnDataSource, LabelSet
from bokeh.plotting import figure, show
from bokeh.io import push_notebook, output_notebook
from bokeh.models import ColumnDataSource, LabelSet
import pandas as pd
```

```
def interactive_tsne(text_labels, tsne_array):
    '''makes an interactive scatter plot with text labels for each point'''

    # Define a dataframe to be used by bokeh context
    bokeh_df = pd.DataFrame(tsne_array, text_labels, columns=['x','y'])
    bokeh_df['text_labels'] = bokeh_df.index
```

(continues on next page)

(continued from previous page)

```
    # interactive controls to include to the plot
    TOOLS="hover, zoom_in, zoom_out, box_zoom, undo, redo, reset, box_select"

    p = figure(tools=TOOLS, plot_width=700, plot_height=700)

    # define data source for the plot
    source = ColumnDataSource(bokeh_df)

    # scatter plot
    p.scatter('x', 'y', source=source, fill_alpha=0.6,
              fill_color="#8724B5",
              line_color=None)

    # text labels
    labels = LabelSet(x='x', y='y', text='text_labels', y_offset=8,
                      text_font_size="8pt", text_color="#555555",
                      source=source, text_align='center')

    p.add_layout(labels)

    # show plot inline
    output_notebook()
    show(p)
```

```
vocab = ['math', 'computing', 'physics']
input_vocab =  [word for word in vocab if word in wv.key_to_index.keys()]
X = wv[input_vocab]
# find tsne coords for 2 dimensions
tsne = TSNE(n_components=2, random_state=0)
X_tsne = tsne.fit_transform(X)

print(input_vocab)

points = len(input_vocab)
interactive_tsne(list(input_vocab)[:points], X_tsne)
```

```
['math', 'computing', 'physics']
```

7.4.2 Training your own Word2Vec Embeddings

Data Crawling from the Web

```
from __future__ import division, unicode_literals
import codecs
from bs4 import BeautifulSoup
import urllib

f = urllib.request.urlopen("https://en.wikipedia.org/wiki/Natural_language_processing
 ↪")

document= BeautifulSoup(f.read()).get_text()
print(document)
```

```
Natural language processing - Wikipedia



Natural language processing

From Wikipedia, the free encyclopedia



Jump to navigation
Jump to search
This article is about natural language processing done by computers. For the natural␣
 ↪language processing done by the human brain, see Language processing in the brain.
Field of computer science and linguistics
 An automated online assistant providing customer service on a web page, an example␣
 ↪of an application where natural language processing is a major component.[1]
Natural language processing (NLP) is a subfield of linguistics, computer science, and␣
 ↪artificial intelligence concerned with the interactions between computers and human␣
 ↪language, in particular how to program computers to process and analyze large␣
 ↪amounts of natural language data.  The goal is a computer capable of "understanding
 ↪" the contents of documents, including the contextual nuances of the language␣
 ↪within them. The technology can then accurately extract information and insights␣
 ↪contained in the documents as well as categorize and organize the documents␣
 ↪themselves.
Challenges in natural language processing frequently involve speech recognition,␣
 ↪natural language understanding, and natural language generation.
```

(continues on next page)

(continued from previous page)

```
Contents

1 History

1.1 Symbolic NLP (1950s - early 1990s)
1.2 Statistical NLP (1990s-2010s)
1.3 Neural NLP (present)


2 Methods: Rules, statistics, neural networks

2.1 Statistical methods
2.2 Neural networks


3 Common NLP tasks

3.1 Text and speech processing
3.2 Morphological analysis
3.3 Syntactic analysis
3.4 Lexical semantics (of individual words in context)
3.5 Relational semantics (semantics of individual sentences)
3.6 Discourse (semantics beyond individual sentences)
3.7 Higher-level NLP applications


4 General tendencies and (possible) future directions

4.1 Cognition and NLP


5 See also
6 References
7 Further reading


History[edit]
Further information: History of natural language processing
Natural language processing has its roots in the 1950s. Already in 1950, Alan Turing␣
 ↪published an article titled "Computing Machinery and Intelligence" which proposed␣
 ↪what is now called the Turing test as a criterion of intelligence, a task that␣
 ↪involves the automated interpretation and generation of natural language, but at␣
 ↪the time not articulated as a problem separate from artificial intelligence.

Symbolic NLP (1950s - early 1990s)[edit]
The premise of symbolic NLP is well-summarized by John Searle's Chinese room␣
 ↪experiment: Given a collection of rules (e.g., a Chinese phrasebook, with questions␣
 ↪and matching answers), the computer emulates natural language understanding (or␣
 ↪other NLP tasks) by applying those rules to the data it is confronted with.

1950s: The Georgetown experiment in 1954 involved fully automatic translation of more␣
 ↪than sixty Russian sentences into English. The authors claimed that within three or␣
 ↪five years, machine translation would be a solved problem.[2]  However, real␣
 ↪progress was much slower, and after the ALPAC report in 1966, which found that ten-␣
 ↪year-long research had failed to fulfill the expectations, funding for machine␣
 ↪translation was dramatically reduced.  Little further research in machine␣
 ↪translation was conducted until the late 1980s when the first statistical machine␣
 ↪translation systems were developed.
```

(continues on next page)

(continued from previous page)

```
1960s: Some notably successful natural language processing systems developed in the␣
 ↪1960s were SHRDLU, a natural language system working in restricted "blocks worlds"␣
 ↪with restricted vocabularies, and ELIZA, a simulation of a Rogerian psychotherapist,
 ↪ written by Joseph Weizenbaum between 1964 and 1966.  Using almost no information␣
 ↪about human thought or emotion, ELIZA sometimes provided a startlingly human-like␣
 ↪interaction. When the "patient" exceeded the very small knowledge base, ELIZA might␣
 ↪provide a generic response, for example, responding to "My head hurts" with "Why do␣
 ↪you say your head hurts?".
1970s: During the 1970s, many programmers began to write "conceptual ontologies",␣
 ↪which structured real-world information into computer-understandable data. ␣
 ↪Examples are MARGIE (Schank, 1975), SAM (Cullingford, 1978), PAM (Wilensky, 1978),␣
 ↪TaleSpin (Meehan, 1976), QUALM (Lehnert, 1977), Politics (Carbonell, 1979), and␣
 ↪Plot Units (Lehnert 1981).  During this time, the first many chatterbots were␣
 ↪written (e.g., PARRY).
1980s: The 1980s and early 1990s mark the hey-day of symbolic methods in NLP. Focus␣
 ↪areas of the time included research on rule-based parsing (e.g., the development of␣
 ↪HPSG as a computational operationalization of generative grammar), morphology (e.g.,
 ↪ two-level morphology[3]), semantics (e.g., Lesk algorithm), reference (e.g.,␣
 ↪within Centering Theory[4]) and other areas of natural language understanding (e.g.,
 ↪ in the Rhetorical Structure Theory). Other lines of research were continued, e.g.,␣
 ↪the development of chatterbots with Racter and Jabberwacky. An important␣
 ↪development (that eventually led to the statistical turn in the 1990s) was the␣
 ↪rising importance of quantitative evaluation in this period.[5]
Statistical NLP (1990s–2010s)[edit]
Up to the 1980s, most natural language processing systems were based on complex sets␣
 ↪of hand-written rules.  Starting in the late 1980s, however, there was a revolution␣
 ↪in natural language processing with the introduction of machine learning algorithms␣
 ↪for language processing.  This was due to both the steady increase in computational␣
 ↪power (see Moore's law) and the gradual lessening of the dominance of Chomskyan␣
 ↪theories of linguistics (e.g. transformational grammar), whose theoretical␣
 ↪underpinnings discouraged the sort of corpus linguistics that underlies the machine-
 ↪learning approach to language processing.[6]

1990s: Many of the notable early successes on statistical methods in NLP occurred in␣
 ↪the field of machine translation, due especially to work at IBM Research.  These␣
 ↪systems were able to take advantage of existing multilingual textual corpora that␣
 ↪had been produced by the Parliament of Canada and the European Union as a result of␣
 ↪laws calling for the translation of all governmental proceedings into all official␣
 ↪languages of the corresponding systems of government.  However, most other systems␣
 ↪depended on corpora specifically developed for the tasks implemented by these␣
 ↪systems, which was (and often continues to be) a major limitation in the success of␣
 ↪these systems. As a result, a great deal of research has gone into methods of more␣
 ↪effectively learning from limited amounts of data.
2000s: With the growth of the web, increasing amounts of raw (unannotated) language␣
 ↪data has become available since the mid-1990s. Research has thus increasingly␣
 ↪focused on unsupervised and semi-supervised learning algorithms.  Such algorithms␣
 ↪can learn from data that has not been hand-annotated with the desired answers or␣
 ↪using a combination of annotated and non-annotated data.  Generally, this task is␣
 ↪much more difficult than supervised learning, and typically produces less accurate␣
 ↪results for a given amount of input data.  However, there is an enormous amount of␣
 ↪non-annotated data available (including, among other things, the entire content of␣
 ↪the World Wide Web), which can often make up for the inferior results if the␣
 ↪algorithm used has a low enough time complexity to be practical.
Neural NLP (present)[edit]
In the 2010s, representation learning and deep neural network-style machine learning␣
 ↪methods became widespread in natural language processing, due in part to a flurry␣
 ↪of results showing that such techniques[7][8] can achieve state-of-the-art results␣
 ↪in many natural language tasks, for example in language modeling,[9] parsing,␣
 ↪[10][11] and many others. This is increasingly important in medicine and healthcare,
 ↪ where NLP is being used to analyze notes and text in electronic health records␣
 ↪that would otherwise be inaccessible for study when seeking to improve care.[12]
```

(continues on next page)

(continued from previous page)

```
Methods: Rules, statistics, neural networks[edit]
In the early days, many language-processing systems were designed by symbolic methods,
 ↪ i.e., the hand-coding of a set of rules, coupled with a dictionary lookup:[13][14]␣
 ↪such as by writing grammars or devising heuristic rules for stemming.
More recent systems based on machine-learning algorithms have many advantages over␣
 ↪hand-produced rules:

The learning procedures used during machine learning automatically focus on the most␣
 ↪common cases, whereas when writing rules by hand it is often not at all obvious␣
 ↪where the effort should be directed.
Automatic learning procedures can make use of statistical inference algorithms to␣
 ↪produce models that are robust to unfamiliar input (e.g. containing words or␣
 ↪structures that have not been seen before) and to erroneous input (e.g. with␣
 ↪misspelled words or words accidentally omitted). Generally, handling such input␣
 ↪gracefully with handwritten rules, or, more generally, creating systems of␣
 ↪handwritten rules that make soft decisions, is extremely difficult, error-prone and␣
 ↪time-consuming.
Systems based on automatically learning the rules can be made more accurate simply by␣
 ↪supplying more input data. However, systems based on handwritten rules can only be␣
 ↪made more accurate by increasing the complexity of the rules, which is a much more␣
 ↪difficult task. In particular, there is a limit to the complexity of systems based␣
 ↪on handwritten rules, beyond which the systems become more and more unmanageable.␣
 ↪However, creating more data to input to machine-learning systems simply requires a␣
 ↪corresponding increase in the number of man-hours worked, generally without␣
 ↪significant increases in the complexity of the annotation process.
Despite the popularity of machine learning in NLP research, symbolic methods are␣
 ↪still (2020) commonly used:

when the amount of training data is insufficient to successfully apply machine␣
 ↪learning methods, e.g., for the machine translation of low-resource languages such␣
 ↪as provided by the Apertium system,
for preprocessing in NLP pipelines, e.g., tokenization, or
for postprocessing and transforming the output of NLP pipelines, e.g., for knowledge␣
 ↪extraction from syntactic parses.
Statistical methods[edit]
Since the so-called "statistical revolution"[15][16] in the late 1980s and mid-1990s,␣
 ↪much natural language processing research has relied heavily on machine learning.␣
 ↪The machine-learning paradigm calls instead for using statistical inference to␣
 ↪automatically learn such rules through the analysis of large corpora (the plural␣
 ↪form of corpus, is a set of documents, possibly with human or computer annotations)␣
 ↪of typical real-world examples.
Many different classes of machine-learning algorithms have been applied to natural-
 ↪language-processing tasks. These algorithms take as input a large set of "features"␣
 ↪that are generated from the input data. Increasingly, however, research has focused␣
 ↪on statistical models, which make soft, probabilistic decisions based on attaching␣
 ↪real-valued weights to each input feature (complex-valued embeddings,[17] and␣
 ↪neural networks in general have also been proposed, for e.g. speech[18]). Such␣
 ↪models have the advantage that they can express the relative certainty of many␣
 ↪different possible answers rather than only one, producing more reliable results␣
 ↪when such a model is included as a component of a larger system.
Some of the earliest-used machine learning algorithms, such as decision trees,␣
 ↪produced systems of hard if-then rules similar to existing hand-written rules. ␣
 ↪However, part-of-speech tagging introduced the use of hidden Markov models to␣
 ↪natural language processing, and increasingly, research has focused on statistical␣
 ↪models, which make soft, probabilistic decisions based on attaching real-valued␣
 ↪weights to the features making up the input data. The cache language models upon␣
 ↪which many speech recognition systems now rely are examples of such statistical␣
 ↪models.  Such models are generally more robust when given unfamiliar input,␣
 ↪especially input that contains errors (as is very common for real-world data), and␣
 ↪produce more reliable results when integrated into a larger system comprising␣
 ↪multiple subtasks.
```

(continues on next page)

(continued from previous page)

```
Since the neural turn, statistical methods in NLP research have been largely replaced␣
 ↪by neural networks. However, they continue to be relevant for contexts in which␣
 ↪statistical interpretability and transparency is required.

Neural networks[edit]
Further information: Artificial neural network
A major drawback of statistical methods is that they require elaborate feature␣
 ↪engineering. Since 2015,[19] the field has thus largely abandoned statistical␣
 ↪methods and shifted to neural networks for machine learning. Popular techniques␣
 ↪include the use of word embeddings to capture semantic properties of words, and an␣
 ↪increase in end-to-end learning of a higher-level task (e.g., question answering)␣
 ↪instead of relying on a pipeline of separate intermediate tasks (e.g., part-of-
 ↪speech tagging and dependency parsing). In some areas, this shift has entailed␣
 ↪substantial changes in how NLP systems are designed, such that deep neural network-
 ↪based approaches may be viewed as a new paradigm distinct from statistical natural␣
 ↪language processing. For instance, the term neural machine translation (NMT)␣
 ↪emphasizes the fact that deep learning-based approaches to machine translation␣
 ↪directly learn sequence-to-sequence transformations, obviating the need for␣
 ↪intermediate steps such as word alignment and language modeling that was used in␣
 ↪statistical machine translation (SMT). Latest works tend to use non-technical␣
 ↪structure of a given task to build proper neural network.[20]

Common NLP tasks[edit]
The following is a list of some of the most commonly researched tasks in natural␣
 ↪language processing. Some of these tasks have direct real-world applications, while␣
 ↪others more commonly serve as subtasks that are used to aid in solving larger tasks.
Though natural language processing tasks are closely intertwined, they can be␣
 ↪subdivided into categories for convenience. A coarse division is given below.

Text and speech processing[edit]
Optical character recognition (OCR)
Given an image representing printed text, determine the corresponding text.
Speech recognition
Given a sound clip of a person or people speaking, determine the textual␣
 ↪representation of the speech.  This is the opposite of text to speech and is one of␣
 ↪the extremely difficult problems colloquially termed "AI-complete" (see above).  In␣
 ↪natural speech there are hardly any pauses between successive words, and thus␣
 ↪speech segmentation is a necessary subtask of speech recognition (see below). In␣
 ↪most spoken languages, the sounds representing successive letters blend into each␣
 ↪other in a process termed coarticulation, so the conversion of the analog signal to␣
 ↪discrete characters can be a very difficult process. Also, given that words in the␣
 ↪same language are spoken by people with different accents, the speech recognition␣
 ↪software must be able to recognize the wide variety of input as being identical to␣
 ↪each other in terms of its textual equivalent.
Speech segmentation
Given a sound clip of a person or people speaking, separate it into words.  A subtask␣
 ↪of speech recognition and typically grouped with it.
Text-to-speech
Given a text, transform those units and produce a spoken representation. Text-to-
 ↪speech can be used to aid the visually impaired.[21]
Word segmentation (Tokenization)
Separate a chunk of continuous text into separate words. For a language like English,␣
 ↪this is fairly trivial, since words are usually separated by spaces. However, some␣
 ↪written languages like Chinese, Japanese and Thai do not mark word boundaries in␣
 ↪such a fashion, and in those languages text segmentation is a significant task␣
 ↪requiring knowledge of the vocabulary and morphology of words in the language.␣
 ↪Sometimes this process is also used in cases like bag of words (BOW) creation in␣
 ↪data mining.
```

(continues on next page)

(continued from previous page)

```
Morphological analysis[edit]
Lemmatization
The task of removing inflectional endings only and to return the base dictionary form␣
 ↪of a word which is also known as a lemma. Lemmatization is another technique for␣
 ↪reducing words to their normalized form. But in this case, the transformation␣
 ↪actually uses a dictionary to map words to their actual form.[22]
Morphological segmentation
Separate words into individual morphemes and identify the class of the morphemes. The␣
 ↪difficulty of this task depends greatly on the complexity of the morphology (i.e.,␣
 ↪the structure of words) of the language being considered. English has fairly simple␣
 ↪morphology, especially inflectional morphology, and thus it is often possible to␣
 ↪ignore this task entirely and simply model all possible forms of a word (e.g.,
 ↪"open, opens, opened, opening") as separate words. In languages such as Turkish or␣
 ↪Meitei,[23] a highly agglutinated Indian language, however, such an approach is not␣
 ↪possible, as each dictionary entry has thousands of possible word forms.
Part-of-speech tagging
Given a sentence, determine the part of speech (POS) for each word. Many words,␣
 ↪especially common ones, can serve as multiple parts of speech. For example, "book"␣
 ↪can be a noun ("the book on the table") or verb ("to book a flight"); "set" can be␣
 ↪a noun, verb or adjective; and "out" can be any of at least five different parts of␣
 ↪speech.
Stemming
The process of reducing inflected (or sometimes derived) words to a base form (e.g.,
 ↪"close" will be the root for "closed", "closing", "close", "closer" etc.). Stemming␣
 ↪yields similar results as lemmatization, but does so on grounds of rules, not a␣
 ↪dictionary.
Syntactic analysis[edit]
Grammar induction[24]
Generate a formal grammar that describes a language's syntax.
Sentence breaking (also known as "sentence boundary disambiguation")
Given a chunk of text, find the sentence boundaries. Sentence boundaries are often␣
 ↪marked by periods or other punctuation marks, but these same characters can serve␣
 ↪other purposes (e.g., marking abbreviations).
Parsing
Determine the parse tree (grammatical analysis) of a given sentence. The grammar for␣
 ↪natural languages is ambiguous and typical sentences have multiple possible␣
 ↪analyses: perhaps surprisingly, for a typical sentence there may be thousands of␣
 ↪potential parses (most of which will seem completely nonsensical to a human). There␣
 ↪are two primary types of parsing: dependency parsing and constituency parsing.␣
 ↪Dependency parsing focuses on the relationships between words in a sentence␣
 ↪(marking things like primary objects and predicates), whereas constituency parsing␣
 ↪focuses on building out the parse tree using a probabilistic context-free grammar␣
 ↪(PCFG) (see also stochastic grammar).
Lexical semantics (of individual words in context)[edit]
Lexical semantics
What is the computational meaning of individual words in context?
Distributional semantics
How can we learn semantic representations from data?
Named entity recognition (NER)
Given a stream of text, determine which items in the text map to proper names, such␣
 ↪as people or places, and what the type of each such name is (e.g. person, location,␣
 ↪organization). Although capitalization can aid in recognizing named entities in␣
 ↪languages such as English, this information cannot aid in determining the type of␣
 ↪named entity, and in any case, is often inaccurate or insufficient.  For example,␣
 ↪the first letter of a sentence is also capitalized, and named entities often span␣
 ↪several words, only some of which are capitalized.  Furthermore, many other␣
 ↪languages in non-Western scripts (e.g. Chinese or Arabic) do not have any␣
 ↪capitalization at all, and even languages with capitalization may not consistently␣
 ↪use it to distinguish names. For example, German capitalizes all nouns, regardless␣
 ↪of whether they are names, and French and Spanish do not capitalize names that␣
 ↪serve as adjectives.
```

(continues on next page)

(continued from previous page)

```
Sentiment analysis (see also Multimodal sentiment analysis)
Extract subjective information usually from a set of documents, often using online␣
 ↪reviews to determine "polarity" about specific objects. It is especially useful for␣
 ↪identifying trends of public opinion in social media, for marketing.
Terminology extraction
The goal of terminology extraction is to automatically extract relevant terms from a␣
 ↪given corpus.
Word sense disambiguation
Many words have more than one meaning; we have to select the meaning which makes the␣
 ↪most sense in context.  For this problem, we are typically given a list of words␣
 ↪and associated word senses, e.g. from a dictionary or an online resource such as␣
 ↪WordNet.
Relational semantics (semantics of individual sentences)[edit]
Relationship extraction
Given a chunk of text, identify the relationships among named entities (e.g. who is␣
 ↪married to whom).
Semantic parsing
Given a piece of text (typically a sentence), produce a formal representation of its␣
 ↪semantics, either as a graph (e.g., in AMR parsing) or in accordance with a logical␣
 ↪formalism (e.g., in DRT parsing). This challenge typically includes aspects of␣
 ↪several more elementary NLP tasks from semantics (e.g., semantic role labelling,␣
 ↪word sense disambiguation) and can be extended to include full-fledged discourse␣
 ↪analysis (e.g., discourse analysis, coreference; see Natural language understanding␣
 ↪below).
Semantic role labelling (see also implicit semantic role labelling below)
Given a single sentence, identify and disambiguate semantic predicates (e.g., verbal␣
 ↪frames), then identify and classify the frame elements (semantic roles).
Discourse (semantics beyond individual sentences)[edit]
Coreference resolution
Given a sentence or larger chunk of text, determine which words ("mentions") refer to␣
 ↪the same objects ("entities"). Anaphora resolution is a specific example of this␣
 ↪task, and is specifically concerned with matching up pronouns with the nouns or␣
 ↪names to which they refer. The more general task of coreference resolution also␣
 ↪includes identifying so-called "bridging relationships" involving referring␣
 ↪expressions. For example, in a sentence such as "He entered John's house through␣
 ↪the front door", "the front door" is a referring expression and the bridging␣
 ↪relationship to be identified is the fact that the door being referred to is the␣
 ↪front door of John's house (rather than of some other structure that might also be␣
 ↪referred to).
Discourse analysis
This rubric includes several related tasks.  One task is discourse parsing, i.e.,␣
 ↪identifying the discourse structure of a connected text, i.e. the nature of the␣
 ↪discourse relationships between sentences (e.g. elaboration, explanation, contrast).
 ↪  Another possible task is recognizing and classifying the speech acts in a chunk␣
 ↪of text (e.g. yes-no question, content question, statement, assertion, etc.).
Implicit semantic role labelling
Given a single sentence, identify and disambiguate semantic predicates (e.g., verbal␣
 ↪frames) and their explicit semantic roles in the current sentence (see Semantic␣
 ↪role labelling above). Then, identify semantic roles that are not explicitly␣
 ↪realized in the current sentence, classify them into arguments that are explicitly␣
 ↪realized elsewhere in the text and those that are not specified, and resolve the␣
 ↪former against the local text. A closely related task is zero anaphora resolution,␣
 ↪i.e., the extension of coreference resolution to pro-drop languages.
Recognizing textual entailment
Given two text fragments, determine if one being true entails the other, entails the␣
 ↪other's negation, or allows the other to be either true or false.[25]
Topic segmentation and recognition
```

(continues on next page)

(continued from previous page)

```
Given a chunk of text, separate it into segments each of which is devoted to a topic,␣
 ↪and identify the topic of the segment.
Argument mining
The goal of argument mining is the automatic extraction and identification of␣
 ↪argumentative structures from natural language text with the aid of computer␣
 ↪programs.[26] Such argumentative structures include the premise, conclusions, the␣
 ↪argument scheme and the relationship between the main and subsidiary argument, or␣
 ↪the main and counter-argument within discourse.[27][28]
Higher-level NLP applications[edit]
Automatic summarization (text summarization)
Produce a readable summary of a chunk of text.  Often used to provide summaries of␣
 ↪the text of a known type, such as research papers, articles in the financial␣
 ↪section of a newspaper.
Book generation
Not an NLP task proper but an extension of natural language generation and other NLP␣
 ↪tasks is the creation of full-fledged books. The first machine-generated book was␣
 ↪created by a rule-based system in 1984 (Racter, The policeman's beard is half-
 ↪constructed).[29] The first published work by a neural network was published in␣
 ↪2018, 1 the Road, marketed as a novel, contains sixty million words. Both these␣
 ↪systems are basically elaborate but non-sensical (semantics-free) language models.␣
 ↪The first machine-generated science book was published in 2019 (Beta Writer,␣
 ↪Lithium-Ion Batteries, Springer, Cham).[30] Unlike Racter and 1 the Road, this is␣
 ↪grounded on factual knowledge and based on text summarization.
Dialogue management
Computer systems intended to converse with a human.
Document AI
A Document AI platform sits on top of the NLP technology enabling users with no prior␣
 ↪experience of artificial intelligence, machine learning or NLP to quickly train a␣
 ↪computer to extract the specific data they need from different document types. NLP-
 ↪powered Document AI enables non-technical teams to quickly access information␣
 ↪hidden in documents, for example, lawyers, business analysts and accountants.[31]
Grammatical error correction
Grammatical error detection and correction involves a great band-width of problems on␣
 ↪all levels of linguistic analysis (phonology/orthography, morphology, syntax,␣
 ↪semantics, pragmatics). Grammatical error correction is impactful since it affects␣
 ↪hundreds of millions of people that use or acquire English as a second language. It␣
 ↪has thus been subject to a number of shared tasks since 2011.[32][33][34] As far as␣
 ↪orthography, morphology, syntax and certain aspects of semantics are concerned, and␣
 ↪due to the development of powerful neural language models such as GPT-2, this can␣
 ↪now (2019) be considered a largely solved problem and is being marketed in various␣
 ↪commercial applications.
Machine translation
Automatically translate text from one human language to another.  This is one of the␣
 ↪most difficult problems, and is a member of a class of problems colloquially termed
 ↪"AI-complete", i.e. requiring all of the different types of knowledge that humans␣
 ↪possess (grammar, semantics, facts about the real world, etc.) to solve properly.
Natural language generation (NLG):
Convert information from computer databases or semantic intents into readable human␣
 ↪language.
Natural language understanding (NLU)
Convert chunks of text into more formal representations such as first-order logic␣
 ↪structures that are easier for computer programs to manipulate. Natural language␣
 ↪understanding involves the identification of the intended semantic from the␣
 ↪multiple possible semantics which can be derived from a natural language expression␣
 ↪which usually takes the form of organized notations of natural language concepts.␣
 ↪Introduction and creation of language metamodel and ontology are efficient however␣
 ↪empirical solutions. An explicit formalization of natural language semantics␣
 ↪without confusions with implicit assumptions such as closed-world assumption (CWA)␣
 ↪vs. open-world assumption, or subjective Yes/No vs. objective True/False is␣
 ↪expected for the construction of a basis of semantics formalization.[35]
```

(continues on next page)

(continued from previous page)

```
Question answering
Given a human-language question, determine its answer. Typical questions have a␣
 ↪specific right answer (such as "What is the capital of Canada?"), but sometimes␣
 ↪open-ended questions are also considered (such as "What is the meaning of life?").
General tendencies and (possible) future directions[edit]
Based on long-standing trends in the field, it is possible to extrapolate future␣
 ↪directions of NLP. As of 2020, three trends among the topics of the long-standing␣
 ↪series of CoNLL Shared Tasks can be observed:[36]

Interest on increasingly abstract, "cognitive" aspects of natural language (1999-
 ↪2001: shallow parsing, 2002-03: named entity recognition, 2006-09/2017-18:␣
 ↪dependency syntax, 2004-05/2008-09 semantic role labelling, 2011-12 coreference,␣
 ↪2015-16: discourse parsing, 2019: semantic parsing).
Increasing interest in multilinguality, and, potentially, multimodality (English␣
 ↪since 1999; Spanish, Dutch since 2002; German since 2003; Bulgarian, Danish,␣
 ↪Japanese, Portuguese, Slovenian, Swedish, Turkish since 2006; Basque, Catalan,␣
 ↪Chinese, Greek, Hungarian, Italian, Turkish since 2007; Czech since 2009; Arabic␣
 ↪since 2012; 2017: 40+ languages; 2018: 60+/100+ languages)
Elimination of symbolic representations (rule-based over supervised towards weakly␣
 ↪supervised methods, representation learning and end-to-end systems)
Cognition and NLP[edit]
Most higher-level NLP applications involve aspects that emulate intelligent behaviour␣
 ↪and apparent comprehension of natural language. More broadly speaking, the␣
 ↪technical operationalization of increasingly advanced aspects of cognitive␣
 ↪behaviour represents one of the developmental trajectories of NLP (see trends among␣
 ↪CoNLL shared tasks above).
Cognition refers to "the mental action or process of acquiring knowledge and␣
 ↪understanding through thought, experience, and the senses."[37] Cognitive science␣
 ↪is the interdisciplinary, scientific study of the mind and its processes.[38]␣
 ↪Cognitive linguistics is an interdisciplinary branch of linguistics, combining␣
 ↪knowledge and research from both psychology and linguistics.[39] Especially during␣
 ↪the age of symbolic NLP, the area of computational linguistics maintained strong␣
 ↪ties with cognitive studies.
As an example, George Lakoff offers a methodology to build natural language␣
 ↪processing (NLP) algorithms through the perspective of cognitive science, along␣
 ↪with the findings of cognitive linguistics,[40] with two defining aspects:

Apply the theory of conceptual metaphor, explained by Lakoff as “the understanding of␣
 ↪one idea, in terms of another” which provides an idea of the intent of the author.
 ↪[41] For example, consider the English word “big”. When used in a comparison (“That␣
 ↪is a big tree”), the author's intent is to imply that the tree is ”physically␣
 ↪large” relative to other trees or the authors experience.  When used metaphorically␣
 ↪(”Tomorrow is a big day”), the author’s intent to imply ”importance”.  The intent␣
 ↪behind other usages, like in ”She is a big person” will remain somewhat ambiguous␣
 ↪to a person and a cognitive NLP algorithm alike without additional information.
Assign relative measures of meaning to a word, phrase, sentence or piece of text␣
 ↪based on the information presented before and after the piece of text being␣
 ↪analyzed, e.g., by means of a probabilistic context-free grammar (PCFG). The␣
 ↪mathematical equation for such algorithms is presented in US patent 9269353 :




R
M
M
```

(continues on next page)

(continued from previous page)

$$
\begin{aligned}
&(token_N) = PMM(token_N) \times \frac{1}{2d} \\
&\left(\sum_{i=-d} d \right)
\end{aligned}
$$

(continues on next page)

(continued from previous page)

```
(
(
P
M
M
(
t
o
k
e

n

N
-
1


)

x

P
F
(
t
o
k
e

n

N


,
t
o
k
e

n

N
-
1


)

)

i
```

(continues on next page)

(continued from previous page)

```
)



{\displaystyle {RMM(token_{N})}={PMM(token_{N})}\times {\frac {1}{2d}}\left(\sum _{i=-
↪d}^{d}{((PMM(token_{N-1}))}\times {PF(token_{N},token_{N-1}))_{i}}\right)}


Where,
RMM, is the Relative Measure of Meaning
token, is any block of text, sentence, phrase or word
N, is the number of tokens being analyzed
PMM, is the Probable Measure of Meaning based on a corpora
d, is the location of the token along the sequence of N-1 tokens
PF, is the Probability Function specific to a language
Ties with cognitive linguistics are part of the historical heritage of NLP, but they␣
↪have been less frequently addressed since the statistical turn during the 1990s.␣
↪Nevertheless, approaches to develop cognitive models towards technically␣
↪operationalizable frameworks have been pursued in the context of various frameworks,
↪ e.g., of cognitive grammar,[42] functional grammar,[43] construction grammar,[44]␣
↪computational psycholinguistics and cognitive neuroscience (e.g., ACT-R), however,␣
↪with limited uptake in mainstream NLP (as measured by presence on major␣
↪conferences[45] of the ACL). More recently, ideas of cognitive NLP have been␣
↪revived as an approach to achieve explainability, e.g., under the notion of
↪"cognitive AI".[46] Likewise, ideas of cognitive NLP are inherent to neural models␣
↪multimodal NLP (although rarely made explicit).[47]

See also[edit]

1 the Road
Automated essay scoring
Biomedical text mining
Compound term processing
Computational linguistics
Computer-assisted reviewing
Controlled natural language
Deep learning
Deep linguistic processing
Distributional semantics
Foreign language reading aid
Foreign language writing aid
Information extraction
Information retrieval
Language and Communication Technologies
Language technology
Latent semantic indexing
Native-language identification
Natural language programming
Natural language search
Outline of natural language processing
Query expansion
Query understanding
Reification (linguistics)
Speech processing
Spoken dialogue systems
Text-proofing
Text simplification
```

(continues on next page)

(continued from previous page)

```
Transformer (machine learning model)
Truecasing
Question answering
Word2vec

References[edit]


^ Kongthon, Alisa; Sangkeettrakarn, Chatchawal; Kongyoung, Sarawoot; Haruechaiyasak,␣
↪Choochart (October 27–30, 2009). Implementing an online help desk system based on␣
↪conversational agent. MEDES '09: The International Conference on Management of␣
↪Emergent Digital EcoSystems. France: ACM. doi:10.1145/1643823.1643908.

^ Hutchins, J. (2005). "The history of machine translation in a nutshell" (PDF).[self-
↪published source]

^ Koskenniemi, Kimmo (1983), Two-level morphology: A general computational model of␣
↪word-form recognition and production (PDF), Department of General Linguistics,␣
↪University of Helsinki

^ Joshi, A. K., & Weinstein, S. (1981, August). Control of Inference: Role of Some␣
↪Aspects of Discourse Structure-Centering. In IJCAI (pp. 385–387).

^ Guida, G.; Mauri, G. (July 1986). "Evaluation of natural language processing␣
↪systems: Issues and approaches". Proceedings of the IEEE. 74 (7): 1026–1035. doi:10.
↪1109/PROC.1986.13580. ISSN 1558-2256. S2CID 30688575.

^ Chomskyan linguistics encourages the investigation of "corner cases" that stress␣
↪the limits of its theoretical models (comparable to pathological phenomena in␣
↪mathematics), typically created using thought experiments, rather than the␣
↪systematic investigation of typical phenomena that occur in real-world data, as is␣
↪the case in corpus linguistics.  The creation and use of such corpora of real-world␣
↪data is a fundamental part of machine-learning algorithms for natural language␣
↪processing.  In addition, theoretical underpinnings of Chomskyan linguistics such␣
↪as the so-called "poverty of the stimulus" argument entail that general learning␣
↪algorithms, as are typically used in machine learning, cannot be successful in␣
↪language processing.  As a result, the Chomskyan paradigm discouraged the␣
↪application of such models to language processing.

^ Goldberg, Yoav (2016). "A Primer on Neural Network Models for Natural Language␣
↪Processing". Journal of Artificial Intelligence Research. 57: 345–420. arXiv:1807.
↪10854. doi:10.1613/jair.4992. S2CID 8273530.

^ Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). Deep Learning. MIT Press.

^ Jozefowicz, Rafal; Vinyals, Oriol; Schuster, Mike; Shazeer, Noam; Wu, Yonghui␣
↪(2016). Exploring the Limits of Language Modeling. arXiv:1602.02410.␣
↪Bibcode:2016arXiv160202410J.

^ Choe, Do Kook; Charniak, Eugene. "Parsing as Language Modeling". Emnlp 2016.

^ Vinyals, Oriol;  et al. (2014). "Grammar as a Foreign Language" (PDF). Nips2015.␣
↪arXiv:1412.7449. Bibcode:2014arXiv1412.7449V.

^ Turchin, Alexander; Florez Builes, Luisa F. (2021-03-19). "Using Natural Language␣
↪Processing to Measure and Improve Quality of Diabetes Care: A Systematic Review".␣
↪Journal of Diabetes Science and Technology. 15 (3): 553–560. doi:10.1177/
↪19322968211000831. ISSN 1932-2968. PMC 8120048. PMID 33736486.
```

(continues on next page)

(continued from previous page)


```
^ Winograd, Terry (1971). Procedures as a Representation for Data in a Computer␣
↪Program for Understanding Natural Language (Thesis).

^ Schank, Roger C.; Abelson, Robert P. (1977). Scripts, Plans, Goals, and␣
↪Understanding: An Inquiry Into Human Knowledge Structures. Hillsdale: Erlbaum. ISBN␣
↪0-470-99033-3.

^ Mark Johnson. How the statistical revolution changes (computational) linguistics.␣
↪Proceedings of the EACL 2009 Workshop on the Interaction between Linguistics and␣
↪Computational Linguistics.

^ Philip Resnik. Four revolutions. Language Log, February 5, 2011.

^ "Investigating complex-valued representation in NLP" (PDF).

^ Trabelsi, Chiheb; Bilaniuk, Olexa; Zhang, Ying; Serdyuk, Dmitriy; Subramanian,␣
↪Sandeep; Santos, João Felipe; Mehri, Soroush; Rostamzadeh, Negar; Bengio, Yoshua;␣
↪Pal, Christopher J. (2018-02-25). "Deep Complex Networks". arXiv:1705.09792 [cs.NE].

^ Socher, Richard. "Deep Learning For NLP-ACL 2012 Tutorial". www.socher.org.␣
↪Retrieved 2020-08-17. This was an early Deep Learning tutorial at the ACL 2012 and␣
↪met with both interest and (at the time) skepticism by most participants. Until␣
↪then, neural learning was basically rejected because of its lack of statistical␣
↪interpretability. Until 2015, deep learning had evolved into the major framework of␣
↪NLP.

^ Annamoradnejad, I. and Zoghi, G. (2020). Colbert: Using bert sentence embedding for␣
↪humor detection. arXiv preprint arXiv:2004.12765.

^ Yi, Chucai; Tian, Yingli (2012), "Assistive Text Reading from Complex Background␣
↪for Blind Persons", Camera-Based Document Analysis and Recognition, Springer Berlin␣
↪Heidelberg, pp. 15-28, CiteSeerX 10.1.1.668.869, doi:10.1007/978-3-642-29364-1_2,␣
↪ISBN 9783642293634

^ "What is Natural Language Processing? Intro to NLP in Machine Learning". GyanSetu!.␣
↪2020-12-06. Retrieved 2021-01-09.

^ Kishorjit, N.; Vidya, Raj RK.; Nirmal, Y.; Sivaji, B. (2012). "Manipuri Morpheme␣
↪Identification" (PDF). Proceedings of the 3rd Workshop on South and Southeast Asian␣
↪Natural Language Processing (SANLP). COLING 2012, Mumbai, December 2012: 95-108.CS1␣
↪maint: location (link)

^ Klein, Dan; Manning, Christopher D. (2002). "Natural language grammar induction␣
↪using a constituent-context model" (PDF). Advances in Neural Information Processing␣
↪Systems.

^ PASCAL Recognizing Textual Entailment Challenge (RTE-7) https://tac.nist.gov//2011/
↪RTE/

^ Lippi, Marco; Torroni, Paolo (2016-04-20). "Argumentation Mining: State of the Art␣
↪and Emerging Trends". ACM Transactions on Internet Technology. 16 (2): 1-25. doi:10.
↪1145/2850417. ISSN 1533-5399. S2CID 9561587.

^ "Argument Mining - IJCAI2016 Tutorial". www.i3s.unice.fr. Retrieved 2021-03-09.

^ "NLP Approaches to Computational Argumentation - ACL 2016, Berlin". Retrieved 2021-
↪03-09.
```


(continues on next page)

(continued from previous page)

^ "U B U W E B :: Racter". www.ubu.com. Retrieved 2020-08-17.

^ Writer, Beta (2019). Lithium-Ion Batteries. doi:10.1007/978-3-030-16800-1. ISBN 978-3-030-16799-8.

^ "Document Understanding AI on Google Cloud (Cloud Next '19) - YouTube". www.youtube.com. Retrieved 2021-01-11.

^ Administration. "Centre for Language Technology (CLT)". Macquarie University. Retrieved 2021-01-11.

^ "Shared Task: Grammatical Error Correction". www.comp.nus.edu.sg. Retrieved 2021-01-11.

^ "Shared Task: Grammatical Error Correction". www.comp.nus.edu.sg. Retrieved 2021-01-11.

^ Duan, Yucong; Cruz, Christophe (2011). "Formalizing Semantic of Natural Language through Conceptualization from Existence". International Journal of Innovation, Management and Technology. 2 (1): 37-42. Archived from the original on 2011-10-09.

^ "Previous shared tasks | CoNLL". www.conll.org. Retrieved 2021-01-11.

^ "Cognition". Lexico. Oxford University Press and Dictionary.com. Retrieved 6 May 2020.

^ "Ask the Cognitive Scientist". American Federation of Teachers. 8 August 2014. Cognitive science is an interdisciplinary field of researchers from Linguistics, psychology, neuroscience, philosophy, computer science, and anthropology that seek to understand the mind.

^ Robinson, Peter (2008). Handbook of Cognitive Linguistics and Second Language Acquisition. Routledge. pp. 3-8. ISBN 978-0-805-85352-0.

^ Lakoff, George (1999). Philosophy in the Flesh: The Embodied Mind and Its Challenge to Western Philosophy; Appendix: The Neural Theory of Language Paradigm. New York Basic Books. pp. 569-583. ISBN 978-0-465-05674-3.

^ Strauss, Claudia (1999). A Cognitive Theory of Cultural Meaning. Cambridge University Press. pp. 156-164. ISBN 978-0-521-59541-4.

^ "Universal Conceptual Cognitive Annotation (UCCA)". Universal Conceptual Cognitive Annotation (UCCA). Retrieved 2021-01-11.

^ Rodríguez, F. C., & Mairal-Usón, R. (2016). Building an RRG computational grammar. Onomazein, (34), 86-117.

^ "Fluid Construction Grammar - A fully operational processing system for construction grammars". Retrieved 2021-01-11.

^ "ACL Member Portal | The Association for Computational Linguistics Member Portal". www.aclweb.org. Retrieved 2021-01-11.

^ "Chunks and Rules". www.w3.org. Retrieved 2021-01-11.

^ Socher, Richard; Karpathy, Andrej; Le, Quoc V.; Manning, Christopher D.; Ng, Andrew Y. (2014). "Grounded Compositional Semantics for Finding and Describing Images with Sentences". Transactions of the Association for Computational Linguistics. 2: 207-218. doi:10.1162/tacl_a_00177. S2CID 2317858.

(continues on next page)

(continued from previous page)

```
Further reading[edit]

Bates, M (1995). "Models of natural language understanding". Proceedings of the␣
 ↪National Academy of Sciences of the United States of America. 92 (22): 9977-9982.␣
 ↪Bibcode:1995PNAS...92.9977B. doi:10.1073/pnas.92.22.9977. PMC 40721. PMID 7479812.
Steven Bird, Ewan Klein, and Edward Loper (2009). Natural Language Processing with␣
 ↪Python. O'Reilly Media. ISBN 978-0-596-51649-9.
Daniel Jurafsky and James H. Martin (2008). Speech and Language Processing, 2nd␣
 ↪edition. Pearson Prentice Hall. ISBN 978-0-13-187321-6.
Mohamed Zakaria Kurdi (2016). Natural Language Processing and Computational␣
 ↪Linguistics: speech, morphology, and syntax, Volume 1. ISTE-Wiley. ISBN 978-
 ↪1848218482.
Mohamed Zakaria Kurdi (2017). Natural Language Processing and Computational␣
 ↪Linguistics: semantics, discourse, and applications, Volume 2. ISTE-Wiley. ISBN 978-
 ↪1848219212.
Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze (2008). Introduction␣
 ↪to Information Retrieval. Cambridge University Press. ISBN 978-0-521-86571-5.␣
 ↪Official html and pdf versions available without charge.
Christopher D. Manning and Hinrich Schütze (1999). Foundations of Statistical Natural␣
 ↪Language Processing. The MIT Press. ISBN 978-0-262-13360-9.
David M. W. Powers and Christopher C. R. Turk (1989). Machine Learning of Natural␣
 ↪Language. Springer-Verlag. ISBN 978-0-387-19557-5.



Wikimedia Commons has media related to Natural language processing.

vteNatural language processingGeneral terms
AI-complete
Bag-of-words
n-gram
Bigram
Trigram
Computational linguistics
Natural-language understanding
Stopwords
Text processing
Text analysis
Collocation extraction
Concept mining
Coreference resolution
Deep linguistic processing
Distant reading
Information extraction
Named-entity recognition
Ontology learning
Parsing
Part-of-speech tagging
Semantic role labeling
Semantic similarity
Sentiment analysis
Terminology extraction
Text mining
Textual entailment
```

(continues on next page)

(continued from previous page)

```
Truecasing
Word-sense disambiguation
Word-sense induction
Text segmentation
Compound-term processing
Lemmatisation
Lexical analysis
Text chunking
Stemming
Sentence segmentation
Word segmentation

Automatic summarization
Multi-document summarization
Sentence extraction
Text simplification
Machine translation
Computer-assisted
Example-based
Rule-based
Statistical
Transfer-based
Neural
Distributional semantics models
BERT
Document-term matrix
Explicit semantic analysis
fastText
GloVe
Latent semantic analysis
Word embedding
Word2vec
Language resources, datasets and corporaTypes and standards
Corpus linguistics
Lexical resource
Linguistic Linked Open Data
Machine-readable dictionary
Parallel text
PropBank
Semantic network
Simple Knowledge Organization System
Speech corpus
Text corpus
Thesaurus (information retrieval)
Treebank
Universal Dependencies
Data
BabelNet
Bank of English
DBpedia
FrameNet
Google Ngram Viewer
ThoughtTreasure
UBY
WordNet
Automatic identificationand data capture
Speech recognition
```

(continues on next page)

(continued from previous page)

```
Speech segmentation
Speech synthesis
Natural language generation
Optical character recognition
Topic model
Document classification
Latent Dirichlet allocation
Pachinko allocation
Computer-assistedreviewing
Automated essay scoring
Concordancer
Grammar checker
Predictive text
Spell checker
Syntax guessing
Natural languageuser interface
Chatbot
Interactive fiction
Question answering
Virtual assistant
Voice user interface
Other software
Natural Language Toolkit
spaCy

Authority control: National libraries
United States
Japan

Language portal




Retrieved from "https://en.wikipedia.org/w/index.php?title=Natural_language_
 ↪processing&oldid=1037947074"
Categories: Natural language processingComputational linguisticsSpeech␣
 ↪recognitionComputational fields of studyArtificial intelligenceHidden categories:␣
 ↪CS1 maint: locationArticles with short descriptionShort description matches␣
 ↪WikidataCommons category link from WikidataArticles with LCCN identifiersArticles␣
 ↪with NDL identifiers







Navigation menu


Personal tools


Not logged inTalkContributionsCreate accountLog in
```

(continues on next page)

(continued from previous page)

```
Namespaces

ArticleTalk



Variants









Views

ReadEditView history



More









Search

















Navigation
```

(continues on next page)

(continued from previous page)

```
Main pageContentsCurrent eventsRandom articleAbout WikipediaContact usDonate



Contribute


HelpLearn to editCommunity portalRecent changesUpload file



Tools


What links hereRelated changesUpload fileSpecial pagesPermanent linkPage⌴
 ↪informationCite this pageWikidata item



Print/export


Download as PDFPrintable version



In other projects


Wikimedia Commons



Languages


Afrikaansةيبرعل|Azərbaycanca□□□□□Bân-lâm-gúБеларускаяБеларуская⌴
 ↪(тарашкевіца)БългарскиCatalàČeštinaDanskDeutschEestiΕλληνικάEspañolEuskaraفارسی|FrançaisGalego□□□ʔա
 ↪IndonesiaÍslenskaItalianoעברית□□□□□ქართულიLietuviųМакедонски□□□□□Монгол□□□□□□□□□□□□□□□□□□□Piemontèis
 ↪Englishاردوکرد|Српски / srpskiSrpskohrvatski /⌴
 ↪српскохрватскиSuomi□□□□□□□□□TürkçeУкраїнськаTiếng Việt□□□□
Edit links




 This page was last edited on 9 August 2021, at 16:32 (UTC).
Text is available under the Creative Commons Attribution-ShareAlike License;
additional terms may apply.  By using this site, you agree to the Terms of Use and⌴
 ↪Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation,⌴
 ↪Inc., a non-profit organization.


Privacy policy
About Wikipedia
```

(continues on next page)

(continued from previous page)

```
Disclaimers
Contact Wikipedia
Mobile view
Developers
Statistics
Cookie statement
```

Pre-processing

```
import gensim
import re
from gensim.corpora import Dictionary
```

```
doc_tokenized = gensim.utils.simple_preprocess(str(document), deacc=True)
doc_tokenized[:10]
```

```
['natural',
 'language',
 'processing',
 'wikipedia',
 'natural',
 'language',
 'processing',
 'from',
 'wikipedia',
 'the']
```

```
dictionary = Dictionary()
BoW_corpus = dictionary.doc2bow(doc_tokenized, allow_update=True)
BoW_corpus = [(dictionary[id], freq) for id, freq in BoW_corpus]
BoW_corpus[:10]
```

```
[('aaron', 1),
 ('abandoned', 1),
 ('abbreviations', 1),
 ('abelson', 1),
 ('able', 2),
 ('about', 5),
 ('above', 3),
 ('abstract', 1),
 ('academy', 1),
 ('accents', 1)]
```

```
def convert(tup, dict):
    for a, b in tup:
        dict[a] = b
    return dict
BoW_corpus_dict = dict()
convert(BoW_corpus, BoW_corpus_dict)
```

```
{'aaron': 1,
 'abandoned': 1,
```

(continues on next page)

(continued from previous page)

```
 'abbreviations': 1,
 'abelson': 1,
 'able': 2,
 'about': 5,
 'above': 3,
 'abstract': 1,
 'academy': 1,
 'accents': 1,
 'access': 1,
 'accidentally': 1,
 'accordance': 1,
 'accountants': 1,
 'accountlog': 1,
 'accurate': 3,
 'accurately': 1,
 'achieve': 2,
 'acl': 5,
 'aclweb': 1,
 'acm': 2,
 'acquire': 1,
 'acquiring': 1,
 'acquisition': 1,
 'act': 1,
 'action': 1,
 'acts': 1,
 'actual': 1,
 'actually': 1,
 'addition': 1,
 'additional': 2,
 'addressed': 1,
 'adjective': 1,
 'adjectives': 1,
 'administration': 1,
 'advanced': 1,
 'advances': 1,
 'advantage': 2,
 'advantages': 1,
 'affects': 1,
 'after': 2,
 'against': 1,
 'age': 1,
 'agent': 1,
 'agglutinated': 1,
 'agree': 1,
 'ai': 8,
 'aid': 7,
 'al': 1,
 'alan': 1,
 'alexander': 1,
 'algorithm': 3,
 'algorithms': 12,
 'alignment': 1,
 'alike': 1,
 'alisa': 1,
 'all': 8,
 'allocation': 2,
 'allows': 1,
```

(continues on next page)

(continued from previous page)

```
 'almost': 1,
 'along': 2,
 'alpac': 1,
 'already': 1,
 'also': 14,
 'although': 2,
 'ambiguous': 2,
 'america': 1,
 'american': 1,
 'among': 4,
 'amount': 3,
 'amounts': 3,
 'amr': 1,
 'an': 22,
 'analog': 1,
 'analyses': 1,
 'analysis': 18,
 'analysts': 1,
 'analyze': 2,
 'analyzed': 2,
 'anaphora': 2,
 'and': 141,
 'andrej': 1,
 'andrew': 1,
 'annamoradnejad': 1,
 'annotated': 4,
 'annotation': 3,
 'annotations': 1,
 'another': 4,
 'answer': 2,
 'answering': 4,
 'answers': 3,
 'anthropology': 1,
 'any': 5,
 'apertium': 1,
 'apparent': 1,
 'appendix': 1,
 'application': 2,
 'applications': 6,
 'applied': 1,
 'apply': 3,
 'applying': 1,
 'approach': 3,
 'approaches': 5,
 'arabic': 2,
 'archived': 1,
 'are': 28,
 'area': 1,
 'areas': 3,
 'argument': 7,
 'argumentation': 2,
 'argumentative': 2,
 'arguments': 1,
 'art': 2,
 'article': 2,
 'articleabout': 1,
 'articles': 1,
```

(continues on next page)

(continued from previous page)

```
 'articletalk': 1,
 'articulated': 1,
 'artificial': 5,
 'arxiv': 8,
 'as': 53,
 'asian': 1,
 'ask': 1,
 'aspects': 7,
 'assertion': 1,
 'assign': 1,
 'assistant': 2,
 'assisted': 2,
 'assistive': 1,
 'associated': 1,
 'association': 2,
 'assumption': 2,
 'assumptions': 1,
 'at': 8,
 'attaching': 2,
 'attribution': 1,
 'august': 3,
 'author': 3,
 'authority': 1,
 'authors': 2,
 'automated': 4,
 'automatic': 6,
 'automatically': 5,
 'available': 4,
 'babelnet': 1,
 'background': 1,
 'bag': 2,
 'ban': 1,
 'band': 1,
 'bank': 1,
 'base': 3,
 'based': 21,
 'basic': 1,
 'basically': 2,
 'basis': 1,
 'basque': 1,
 'bates': 1,
 'batteries': 2,
 'be': 26,
 'beard': 1,
 'became': 1,
 'because': 1,
 'become': 2,
 'been': 10,
 'before': 2,
 'began': 1,
 'behaviour': 2,
 'behind': 1,
 'being': 8,
 'below': 4,
 'bengio': 2,
 'berlin': 2,
 'bert': 2,
```

(continues on next page)

(continued from previous page)

```
 'beta': 2,
 'between': 7,
 'beyond': 3,
 'bibcode': 3,
 'big': 4,
 'bigram': 1,
 'bilaniuk': 1,
 'biomedical': 1,
 'bird': 1,
 'blend': 1,
 'blind': 1,
 'block': 1,
 'blocks': 1,
 'book': 6,
 'books': 2,
 'both': 4,
 'boundaries': 3,
 'boundary': 1,
 'bow': 1,
 'brain': 2,
 'branch': 1,
 'breaking': 1,
 'bridging': 2,
 'broadly': 1,
 'build': 2,
 'building': 2,
 'builes': 1,
 'bulgarian': 1,
 'business': 1,
 'but': 8,
 'by': 24,
 'cache': 1,
 'called': 4,
 'calling': 1,
 'calls': 1,
 'cambridge': 2,
 'camera': 1,
 'can': 22,
 'canada': 2,
 'cannot': 2,
 'capable': 1,
 'capital': 1,
 'capitalization': 3,
 'capitalize': 1,
 'capitalized': 2,
 'capitalizes': 1,
 'capture': 2,
 'carbonell': 1,
 'care': 2,
 'case': 3,
 'cases': 3,
 'catalan': 1,
 'categories': 3,
 'categorize': 1,
 'category': 1,
 'centering': 2,
 'centre': 1,
```

(continues on next page)

(continued from previous page)

```
 'certain': 1,
 'certainty': 1,
 'challenge': 3,
 'challenges': 1,
 'cham': 1,
 'changes': 2,
 'changesupload': 2,
 'character': 2,
 'characters': 2,
 'charge': 1,
 'charniak': 1,
 'chatbot': 1,
 'chatchawal': 1,
 'chatterbots': 2,
 'checker': 2,
 'chiheb': 1,
 'chinese': 5,
 'choe': 1,
 'chomskyan': 4,
 'choochart': 1,
 'christophe': 1,
 'christopher': 6,
 'chucai': 1,
 'chunk': 7,
 'chunking': 1,
 'chunks': 2,
 'cid': 4,
 'citeseerx': 1,
 'claimed': 1,
 'class': 2,
 'classes': 1,
 'classification': 1,
 'classify': 2,
 'classifying': 1,
 'claudia': 1,
 'clip': 2,
 'close': 2,
 'closed': 2,
 'closely': 2,
 'closer': 1,
 'closing': 1,
 'cloud': 2,
 'clt': 1,
 'coarse': 1,
 'coarticulation': 1,
 'coding': 1,
 'cognition': 4,
 'cognitive': 21,
 'colbert': 1,
 'coling': 1,
 'collection': 1,
 'collocation': 1,
 'colloquially': 2,
 'com': 3,
 'combination': 1,
 'combining': 1,
 'commercial': 1,
```

(continues on next page)

(continued from previous page)

```
 'common': 5,
 'commonly': 3,
 'commons': 3,
 'communication': 1,
 'comp': 2,
 'comparable': 1,
 'comparison': 1,
 'complete': 3,
 'completely': 1,
 'complex': 5,
 'complexity': 5,
 'component': 2,
 'compositional': 1,
 'compound': 2,
 'comprehension': 1,
 'comprising': 1,
 'computational': 16,
 'computer': 16,
 'computers': 3,
 'computing': 1,
 'concept': 1,
 'concepts': 1,
 'conceptual': 4,
 'concerned': 3,
 'conclusions': 1,
 'concordancer': 1,
 'conducted': 1,
 'conference': 1,
 'conferences': 1,
 'confronted': 1,
 'confusions': 1,
 'conll': 4,
 'connected': 1,
 'consider': 1,
 'considered': 3,
 'consistently': 1,
 'constituency': 2,
 'constituent': 1,
 'constructed': 1,
 'construction': 4,
 'consuming': 1,
 'contact': 1,
 'contained': 1,
 'containing': 1,
 'contains': 2,
 'content': 2,
 'contents': 2,
 'context': 8,
 'contexts': 1,
 'contextual': 1,
 'continue': 1,
 'continued': 1,
 'continues': 1,
 'continuous': 1,
 'contrast': 1,
 'contribute': 1,
 'control': 2,
```

(continues on next page)

(continued from previous page)

```
 'controlled': 1,
 'convenience': 1,
 'conversational': 1,
 'converse': 1,
 'conversion': 1,
 'convert': 2,
 'cookie': 1,
 'coreference': 6,
 'corner': 1,
 'corpora': 5,
 'corporatypes': 1,
 'corpus': 7,
 'correction': 5,
 'corresponding': 3,
 'counter': 1,
 'coupled': 1,
 'courville': 1,
 'created': 2,
 'creating': 2,
 'creation': 4,
 'creative': 1,
 'criterion': 1,
 'cruz': 1,
 'cs': 3,
 'cullingford': 1,
 'cultural': 1,
 'current': 2,
 'customer': 1,
 'cwa': 1,
 'czech': 1,
 'dan': 1,
 'daniel': 1,
 'danish': 1,
 'data': 24,
 'databases': 1,
 'datasets': 1,
 'david': 1,
 'day': 2,
 'days': 1,
 'dbpedia': 1,
 'deal': 1,
 'december': 1,
 'decision': 1,
 'decisions': 3,
 'deep': 11,
 'defining': 1,
 'department': 1,
 'depended': 1,
 'dependencies': 1,
 'dependency': 4,
 'depends': 1,
 'derived': 2,
 'describes': 1,
 'describing': 1,
 'description': 1,
 'designed': 2,
 'desired': 1,
```

(continues on next page)

(continued from previous page)

```
 'desk': 1,
 'despite': 1,
 'detection': 2,
 'determine': 9,
 'determining': 1,
 'develop': 1,
 'developed': 3,
 'developers': 1,
 'development': 4,
 'developmental': 1,
 'devising': 1,
 'devoted': 1,
 'diabetes': 2,
 'dialogue': 2,
 'dictionary': 8,
 'different': 6,
 'difficult': 6,
 'difficulty': 1,
 'digital': 1,
 'direct': 1,
 'directed': 1,
 'directions': 3,
 'directly': 1,
 'dirichlet': 1,
 'disambiguate': 2,
 'disambiguation': 4,
 'disclaimers': 1,
 'discouraged': 2,
 'discourse': 12,
 'discrete': 1,
 'displaystyle': 1,
 'distant': 1,
 'distinct': 1,
 'distinguish': 1,
 'distributional': 3,
 'division': 1,
 'dmitriy': 1,
 'do': 5,
 'document': 9,
 'documents': 6,
 'does': 1,
 'doi': 9,
 'dominance': 1,
 'done': 2,
 'door': 4,
 'download': 1,
 'dramatically': 1,
 'drawback': 1,
 'drop': 1,
 'drt': 1,
 'duan': 1,
 'due': 4,
 'during': 5,
 'dutch': 1,
 'each': 7,
 'eacl': 1,
 'earliest': 1,
```

(continues on next page)

(continued from previous page)

```
 'early': 6,
 'easier': 1,
 'ecosystems': 1,
 'edit': 21,
 'editcommunity': 1,
 'edited': 1,
 'edition': 1,
 'edu': 2,
 'edward': 1,
 'effectively': 1,
 'efficient': 1,
 'effort': 1,
 'either': 2,
 'elaborate': 2,
 'elaboration': 1,
 'electronic': 1,
 'elementary': 1,
 'elements': 1,
 'elimination': 1,
 'eliza': 3,
 'elsewhere': 1,
 'embedding': 2,
 'embeddings': 2,
 'embodied': 1,
 'emergent': 1,
 'emerging': 1,
 'emnlp': 1,
 'emotion': 1,
 'emphasizes': 1,
 'empirical': 1,
 'emulate': 1,
 'emulates': 1,
 'en': 1,
 'enables': 1,
 'enabling': 1,
 'encourages': 1,
 'encyclopedia': 1,
 'end': 4,
 'ended': 1,
 'endings': 1,
 'engineering': 1,
 'english': 8,
 'enormous': 1,
 'enough': 1,
 'entail': 1,
 'entailed': 1,
 'entailment': 3,
 'entails': 2,
 'entered': 1,
 'entire': 1,
 'entirely': 1,
 'entities': 4,
 'entity': 4,
 'entry': 1,
 'equation': 1,
 'equivalent': 1,
 'erlbaum': 1,
```

(continues on next page)

(continued from previous page)

```
 'erroneous': 1,
 'error': 6,
 'errors': 1,
 'especially': 6,
 'essay': 2,
 'et': 1,
 'etc': 3,
 'eugene': 1,
 'european': 1,
 'evaluation': 2,
 'even': 1,
 'eventsrandom': 1,
 'eventually': 1,
 'evolved': 1,
 'ewan': 1,
 'example': 12,
 'examples': 3,
 'exceeded': 1,
 'existence': 1,
 'existing': 2,
 'expansion': 1,
 'expectations': 1,
 'expected': 1,
 'experience': 3,
 'experiment': 2,
 'experiments': 1,
 'explainability': 1,
 'explained': 1,
 'explanation': 1,
 'explicit': 4,
 'explicitly': 2,
 'exploring': 1,
 'export': 1,
 'express': 1,
 'expression': 2,
 'expressions': 1,
 'extended': 1,
 'extension': 2,
 'extract': 4,
 'extraction': 10,
 'extrapolate': 1,
 'extremely': 2,
 'fact': 2,
 'facts': 1,
 'factual': 1,
 'failed': 1,
 'fairly': 2,
 'false': 2,
 'far': 1,
 'fashion': 1,
 'fasttext': 1,
 'feature': 2,
 'features': 2,
 'february': 1,
 'federation': 1,
 'felipe': 1,
 'fiction': 1,
```

(continues on next page)

(continued from previous page)

```
 'field': 5,
 'fields': 1,
 'file': 1,
 'filespecial': 1,
 'financial': 1,
 'find': 1,
 'finding': 1,
 'findings': 1,
 'first': 7,
 'five': 2,
 'fledged': 2,
 'flesh': 1,
 'flight': 1,
 'florez': 1,
 'fluid': 1,
 'flurry': 1,
 'focus': 2,
 'focused': 3,
 'focuses': 2,
 'following': 1,
 'for': 53,
 'foreign': 3,
 'form': 7,
 'formal': 3,
 'formalism': 1,
 'formalization': 2,
 'formalizing': 1,
 'former': 1,
 'forms': 2,
 'found': 1,
 'foundation': 1,
 'foundations': 1,
 'four': 1,
 'fr': 1,
 'frac': 1,
 'fragments': 1,
 'frame': 1,
 'framenet': 1,
 'frames': 2,
 'framework': 1,
 'frameworks': 2,
 'france': 1,
 'free': 4,
 'french': 1,
 'frequently': 2,
 'from': 25,
 'front': 3,
 'fulfill': 1,
 'full': 2,
 'fully': 2,
 'function': 1,
 'functional': 1,
 'fundamental': 1,
 'funding': 1,
 'further': 5,
 'furthermore': 1,
 'future': 3,
```

(continues on next page)

(continued from previous page)

```
 'general': 7,
 'generally': 5,
 'generate': 1,
 'generated': 3,
 'generation': 6,
 'generative': 1,
 'generic': 1,
 'george': 2,
 'georgetown': 1,
 'german': 2,
 'given': 24,
 'glove': 1,
 'goal': 3,
 'goals': 1,
 'goldberg': 1,
 'gone': 1,
 'goodfellow': 1,
 'google': 2,
 'gov': 1,
 'government': 1,
 'governmental': 1,
 'gpt': 1,
 'gracefully': 1,
 'gradual': 1,
 'gram': 1,
 'grammar': 17,
 'grammars': 2,
 'grammatical': 6,
 'graph': 1,
 'great': 2,
 'greatly': 1,
 'greek': 1,
 'grounded': 2,
 'grounds': 1,
 'grouped': 1,
 'growth': 1,
 'guessing': 1,
 'guida': 1,
 'gyansetu': 1,
 'had': 3,
 'half': 1,
 'hall': 1,
 'hand': 6,
 'handbook': 1,
 'handling': 1,
 'handwritten': 4,
 'hard': 1,
 'hardly': 1,
 'haruechaiyasak': 1,
 'has': 15,
 'have': 15,
 'he': 1,
 'head': 2,
 'health': 1,
 'healthcare': 1,
 'heavily': 1,
 'heidelberg': 1,
```

(continues on next page)

(continued from previous page)

```
'help': 1,
'helplearn': 1,
'helsinki': 1,
'hererelated': 1,
'heritage': 1,
'heuristic': 1,
'hey': 1,
'hidden': 2,
'higher': 4,
'highly': 1,
'hillsdale': 1,
'hinrich': 2,
'historical': 1,
'history': 5,
'hours': 1,
'house': 2,
'how': 4,
'however': 13,
'hpsg': 1,
'hrvatskibahasa': 1,
'html': 1,
'https': 2,
'human': 11,
'humans': 1,
'humor': 1,
'hundreds': 1,
'hungarian': 1,
'hurts': 2,
'hutchins': 1,
'ian': 1,
'ibm': 1,
'idea': 2,
'ideas': 2,
'identical': 1,
'identification': 4,
'identified': 1,
'identifiers': 1,
'identify': 7,
'identifying': 3,
'ieee': 1,
'if': 3,
'ignore': 1,
'ijcai': 2,
'image': 1,
'images': 1,
'impactful': 1,
'impaired': 1,
'implemented': 1,
'implementing': 1,
'implicit': 3,
'imply': 2,
'importance': 2,
'important': 2,
'improve': 2,
'in': 105,
'inaccessible': 1,
'inaccurate': 1,
```

(continues on next page)

(continued from previous page)

```
 'inc': 1,
 'include': 3,
 'included': 2,
 'includes': 3,
 'including': 2,
 'increase': 3,
 'increases': 1,
 'increasing': 3,
 'increasingly': 6,
 'index': 1,
 'indexing': 1,
 'indian': 1,
 'individual': 8,
 'induction': 3,
 'inference': 3,
 'inferior': 1,
 'inflected': 1,
 'inflectional': 2,
 'information': 17,
 'informationcite': 1,
 'inherent': 1,
 'innovation': 1,
 'input': 13,
 'inquiry': 1,
 'insights': 1,
 'instance': 1,
 'instead': 2,
 'insufficient': 2,
 'integrated': 1,
 'intelligence': 6,
 'intelligent': 1,
 'intended': 2,
 'intent': 4,
 'intents': 1,
 'interaction': 2,
 'interactions': 1,
 'interactive': 1,
 'interest': 3,
 'interface': 2,
 'intermediate': 2,
 'international': 2,
 'internet': 1,
 'interpretation': 1,
 'intertwined': 1,
 'into': 16,
 'intro': 1,
 'introduced': 1,
 'introduction': 3,
 'investigating': 1,
 'investigation': 2,
 'involve': 2,
 'involved': 1,
 'involves': 3,
 'involving': 1,
 'ion': 2,
 'is': 82,
 'isbn': 13,
```

(continues on next page)

(continued from previous page)

```
 'issn': 3,
 'issues': 1,
 'iste': 2,
 'it': 11,
 'italian': 1,
 'item': 1,
 'items': 1,
 'its': 8,
 'jabberwacky': 1,
 'jair': 1,
 'james': 1,
 'japan': 1,
 'japanese': 2,
 'joao': 1,
 'john': 3,
 'johnson': 1,
 'joseph': 1,
 'joshi': 1,
 'journal': 3,
 'jozefowicz': 1,
 'july': 1,
 'jump': 2,
 'jurafsky': 1,
 'karpathy': 1,
 'kimmo': 1,
 'kishorjit': 1,
 'klein': 2,
 'knowledge': 9,
 'known': 3,
 'kongthon': 1,
 'kongyoung': 1,
 'kook': 1,
 'koskenniemi': 1,
 'kurdi': 2,
 'labeling': 1,
 'labelling': 6,
 'lack': 1,
 'lakoff': 3,
 'lam': 1,
 'language': 105,
 'languages': 14,
 'languageuser': 1,
 'large': 4,
 'largely': 3,
 'larger': 4,
 'last': 1,
 'late': 3,
 'latent': 3,
 'latest': 1,
 'law': 1,
 'laws': 1,
 'lawyers': 1,
 'lccn': 1,
 'le': 1,
 'learn': 4,
 'learning': 37,
 'least': 1,
```

(continues on next page)

(continued from previous page)

```
 'led': 1,
 'left': 1,
 'lehnert': 2,
 'lemma': 1,
 'lemmatisation': 1,
 'lemmatization': 3,
 'lesk': 1,
 'less': 2,
 'lessening': 1,
 'letter': 1,
 'letters': 1,
 'level': 6,
 'levels': 1,
 'lexical': 5,
 'lexico': 1,
 'libraries': 1,
 'license': 1,
 'life': 1,
 'like': 6,
 'likewise': 1,
 'limit': 1,
 'limitation': 1,
 'limited': 2,
 'limits': 2,
 'lines': 1,
 'linguistic': 4,
 'linguistics': 27,
 'link': 2,
 'linked': 1,
 'linkpage': 1,
 'links': 2,
 'lippi': 1,
 'list': 2,
 'lithium': 2,
 'little': 1,
 'local': 1,
 'location': 3,
 'log': 1,
 'logged': 1,
 'logic': 1,
 'logical': 1,
 'long': 3,
 'lookup': 1,
 'loper': 1,
 'low': 2,
 'luisa': 1,
 'machine': 34,
 'machinery': 1,
 'macquarie': 1,
 'made': 3,
 'main': 3,
 'mainstream': 1,
 'maint': 2,
 'maintained': 1,
 'mairal': 1,
 'major': 5,
 'make': 5,
```

(continues on next page)

(continued from previous page)

```
 'makes': 1,
 'making': 1,
 'man': 1,
 'management': 3,
 'manipulate': 1,
 'manipuri': 1,
 'manning': 4,
 'many': 13,
 'map': 2,
 'marco': 1,
 'margie': 1,
 'mark': 3,
 'marked': 1,
 'marketed': 2,
 'marketing': 1,
 'marking': 2,
 'markov': 1,
 'marks': 1,
 'married': 1,
 'martin': 1,
 'matches': 1,
 'matching': 2,
 'mathematical': 1,
 'mathematics': 1,
 'matrix': 1,
 'mauri': 1,
 'may': 5,
 'meaning': 8,
 'means': 1,
 'measure': 3,
 'measured': 1,
 'measures': 1,
 'medes': 1,
 'media': 3,
 'medicine': 1,
 'meehan': 1,
 'mehri': 1,
 'meitei': 1,
 'member': 3,
 'mental': 1,
 'mentions': 1,
 'menu': 1,
 'met': 1,
 'metamodel': 1,
 'metaphor': 1,
 'metaphorically': 1,
 'methodology': 1,
 'methods': 15,
 'mid': 2,
 'might': 2,
 'mike': 1,
 'million': 1,
 'millions': 1,
 'mind': 3,
 'mining': 8,
 'misspelled': 1,
 'mit': 2,
```

(continues on next page)

(continued from previous page)

```
 'mobile': 1,
 'model': 6,
 'modeling': 4,
 'models': 17,
 'mohamed': 2,
 'moore': 1,
 'more': 23,
 'morpheme': 1,
 'morphemes': 2,
 'morphological': 3,
 'morphology': 10,
 'most': 10,
 'much': 4,
 'multi': 1,
 'multilingual': 1,
 'multilinguality': 1,
 'multimodal': 2,
 'multimodality': 1,
 'multiple': 4,
 'mumbai': 1,
 'must': 1,
 'my': 1,
 'name': 1,
 'named': 7,
 'names': 5,
 'namespaces': 1,
 'national': 2,
 'native': 1,
 'natural': 66,
 ...}
```

Visualisation of Pre-trained Embeddings

We only look at the non-stopwords that are in the Word2Vec vocabulary.

```
import nltk
from nltk.corpus import stopwords

vocab_sorted = dict(sorted(BoW_corpus_dict.items(), key=lambda item: item[1],␣
 ↪reverse=True))
#print(vocab_sorted)

stopwords = stopwords.words('english')
input_vocab =  [word for word in vocab_sorted if word in wv.key_to_index.keys() and␣
 ↪word not in stopwords]
points = len(input_vocab)
X = wv[input_vocab]
X_tsne = tsne.fit_transform(X[:points])

#print(input_vocab)

#points = len(input_vocab)

interactive_tsne(list(input_vocab)[:points], X_tsne)
```

Training your own Word2Vec

```
from gensim.models import Word2Vec

cores = 16
model = Word2Vec(min_count=1,
                     window=2,
                     vector_size=100,
                     sample=6e-5,
                     alpha=0.03,
                     min_alpha=0.0007,
                     negative=20,
                     workers=cores-1)
```

```
from time import time

t = time()

model.build_vocab([doc_tokenized], progress_per=10)

print('Time to build vocab: {} mins'.format(round((time() - t) / 60, 2)))
```

```
Time to build vocab: 0.0 mins
```

```
t = time()

model.train(doc_tokenized, total_examples=model.corpus_count, epochs=1000, report_
 ↪delay=1)

print('Time to train the model: {} mins'.format(round((time() - t) / 60, 2)))
```

```
Time to train the model: 0.16 mins
```

```
model.wv.key_to_index.keys()
```

```
dict_keys(['the', 'of', 'and', 'to', 'language', 'in', 'is', 'natural', 'for', 'as',
 ↪'processing', 'text', 'nlp', 'on', 'that', 'learning', 'or', 'machine', 'with',
 ↪'such', 'speech', 'words', 'are', 'linguistics', 'be', 'this', 'systems', 'from',
 ↪'statistical', 'semantic', 'semantics', 'data', 'given', 'by', 'neural', 'more', 'an
 ↪', 'can', 'which', 'word', 'based', 'edit', 'sentence', 'cognitive', 'rules', 'other
 ↪', 'analysis', 'tasks', 'parsing', 'models', 'information', 'task', 'grammar',
 ↪'retrieved', 'computer', 'into', 'computational', 'methods', 'have', 'research',
 ↪'recognition', 'understanding', 'has', 'since', 'not', 'languages', 'also', 'was',
 ↪'translation', 'isbn', 'used', 'input', 'however', 'many', 'see', 'algorithms',
 ↪'discourse', 'example', 'it', 'human', 'deep', 'possible', 'been', 'using',
 ↪'segmentation', 'world', 'when', 'morphology', 'extraction', 'most', 'real', 'doi',
 ↪'one', 'often', 'determine', 'document', 'knowledge', 'www', 'some', 'role', 'part',
 ↪ 'but', 'at', 'use', 'separate', 'its', 'dictionary', 'english', 'being', 'they',
 ↪'system', 'symbolic', 'mining', 'science', 'networks', 'arxiv', 'sentences',
 ↪'context', 'meaning', 'individual', 'all', 'ai', 'network', 'aid', 'typically', 'pdf
 ↪', 'technology', 'between', 'results', 'representation', 'chunk', 'process', 'corpus
 ↪', 'form', 'were', 'question', 'first', 'each', 'general', 'argument', 'aspects',
 ↪'what', 'non', 'named', 'these', 'identify', 'structure', 'hand', 'textual',
 ↪'increasingly', 'wikipedia', 'there', 'shared', 'difficult', 'especially', 'like',
 ↪'book', 'time', 'christopher', 'different', 'model', 'intelligence', 'grammatical',
 ↪'generation', 'error', 'early', 'level', 'syntax', 'labelling', 'documents',
 ↪'coreference', 'automatic', 'resolution', 'than', 'terms', 'applications', 'may',
 ↪'acl', 'pp', 'people', 'complex', 'person', 'theory', 'names', 'two', 'university',
 ↪'many', 'approaches', 'make', 'token_', 'correction', 'summarization', 'set',
 ↪'generally', 'automatically', 'produce', 'structures', 'so', 'through', 'specific',
 ↪'thus', 'typical', 'proceedings', 'complexity', 'sense', 'trends', 'corpora', 'press
 ↪', 'org', 'history', 'written', 'lexical', 'further', 'common', 'reading', 'do',
 ↪'published', 'during', 'chinese', 'then', 'field', 'major', 'about', 'artificial',
```

(continues on next page)

(continued from previous page)

Comparing the purposely trained and the pre-trained vectors

We can see that due to the lack of data in training our own embeddings, the semantic information captured by the embeddings are not as meaningful as the the pre-trained ones.

```
# Our trained domain specific embeddings
model.wv.most_similar(positive=["language"])
```

```
[('generated', 0.3286932706832886),
 ('cache', 0.29854971170425415),
 ('charge', 0.27286359667778015),
 ('segmentation', 0.26987093687057495),
 ('assisted', 0.26623496413230896),
 ('drop', 0.2609272599220276),
 ('on', 0.25290459394454956),
 ('successfully', 0.2467595636844635),
 ('any', 0.24194024503231049),
 ('easier', 0.2374984174966812)]
```

```
# Pretrained embeddings
wv.most_similar(positive=["language"])
```

```
[('langauge', 0.747669517993927),
 ('Language', 0.6695358157157898),
 ('languages', 0.6341331601142883),
 ('English', 0.6120712757110596),
 ('CMPB_Spanish', 0.6083105802536011),
 ('nonnative_speakers', 0.6063110828399658),
 ('idiomatic_expressions', 0.5889802575111389),
 ('verb_tenses', 0.5841568112373352),
 ('Kumeyaay_Diegueno', 0.5798824429512024),
 ('dialect', 0.5724599957466125)]
```