**Installation of Software**

* Which should you use?
  + R/RStudio: mostly for users who are good with R and want to keep using RStudio because you've been using it a lot for other classes
    - start a rmarkdown assignment and make the assignment
  + Anaconda/Jupyter/Spyder: you are already use python in this system a lot
    - Start a jupyter notebook and make the assignment
  + Datalore: is for people who are newer at python OR bad at installing stuff on their computer OR have an older computer OR have spaces in your username OR you want to be able to work together with your team mates or me
    - start a new notebook and make the assignment
* Notes:
  + You only need keras and tensorflow if you want to try deep learning in the classification section

**Pycharm**

* Install!
* Click learn
* New course
* In marketplace > introduction to python
* Take a screen when you are done with the sections required (not all due at once)
* Be sure you can see the TERMINAL at the bottom in that screen shot

**Processing Text**

On class assignments, we all turn in the the same code and data source.

**Libraries**

# import just a function

from urllib.request import urlopen

from bs4 import BeautifulSoup

# import a whole library as a new name

import pandas as pd

# import package as its own name

import nltk

# other packages

import re

# impurity function

RE\_SUSPICIOUS = re.compile(r'[&#<>{}\[\]\\]')

def impurity(text, min\_len=10):

"""returns the share of suspicious characters in a text"""

if text == None or len(text) < min\_len:

return 0

else:

return len(RE\_SUSPICIOUS.findall(text))/len(text)

# rest of stuff

import textacy.preprocessing as tprep

def normalize(text):

text = tprep.normalize.hyphenated\_words(text)

text = tprep.normalize.quotation\_marks(text)

text = tprep.normalize.unicode(text)

text = tprep.remove.accents(text)

text = tprep.replace.phone\_numbers(text)

text = tprep.replace.urls(text)

text = tprep.replace.emails(text)

text = tprep.replace.user\_handles(text)

text = tprep.replace.emojis(text)

return text

# install pyspellchecker !!!

from spellchecker import SpellChecker

spell = SpellChecker()

import spacy

nlp = spacy.load("en\_core\_web\_sm")

import textacy

from itertools import chain

from collections import Counter

**Find Text**

* As a class, we will find a text source to analyze. This text source usually will consist of a webpage or other dataset to examine and clean.
* Import the text into your report.
* If the text is one big long string, first break into sentence segments and store it in a Pandas DataFrame.

myurl = "https://www.foxnews.com/sports/patrick-mahomes-fiery-message-win-bills-they-got-what-they-asked-for"

#myurl = "https://www.foxnews.com/lifestyle/newly-elected-school-board-pennsylvania-reclaims-indigenous-mascot-rejects-cancel-culture"

html = urlopen(myurl).read()

soupified = BeautifulSoup(html, "html.parser")

# soupified

# just try get\_text()

try\_text = soupified.get\_text()

try\_text[0:100]

* Regular expressions

# find an exact match for the first time this occurs

text = try\_text[

# everything from the end of this sentence and on

re.search("To access the content, check your email and follow the instructions provided.", try\_text).end():

# now the end

re.search("CLICK HERE TO GET THE FOX NEWS APP", try\_text).start()

]

* Breaking down into sentences

# break down into sentences and put into DF

sentences = nltk.sent\_tokenize(text)

type(sentences)

# convert to dataframe

DF = pd.DataFrame(sentences, columns = ["sentence"])

DF.head()

* We've used:
  + One big string (one variable)
  + A list which uses []
  + Dictionaries {}
  + Tuples ()
  + DataFrame from pandas

**Length for Proposal**

# do this on the full text not broken into sentences

len(nltk.word\_tokenize(text))

# be sure to import nltk in the proposal

**Fix Errors**

* Examine the text for errors or problems by looking at the text.
  + Legit, just look at the text.
  + Looking for any type of "garbage" - dependent on what you are doing.
* Use the “impurity” function from class to examine the text for potential issues.

DF['score'] = DF['sentence'].apply(impurity)

DF

* Remove the noise with the regex function.
* Re-examine the impurity to determine if the data has been mostly cleaned.
  + Not necessary because it looks fine.
* Normalize the rest of the text by using textacy.

DF['clean'] = DF['sentence'].apply(normalize)

DF

* Examine spelling errors in at least one row of the dataset.
  + Any time you have stuff with names, please do not do spelling.
  + Mostly, only do this if you have a specific goals.

# find all the unique tokens

# set is find unique

# nltk.word\_tokenize is break down into words

# " ".join is combine into one long text

# .to\_list() is a function to convert to list

clean\_tokens = set(nltk.word\_tokenize(" ".join(DF['clean'].to\_list())))

# what is wrong?

misspelled = spell.unknown(clean\_tokens)

for word in misspelled:

# what's the word

print(word)

print("\n")

# Get the one `most likely` answer

print(spell.correction(word))

# Get a list of `likely` options

print(spell.candidates(word))

# make a dictionary of the misspelled word and the correction

# use find and replace in re to fix them

Pre-Processing

* Using spacy and textacy, pre-process the text to end up with a list of tokenized lists.

output = []

# only the tagger and lemmatizer

for doc in nlp.pipe(DF['clean'].tolist(), disable=["tok2vec", "ner", "parser"]):

tokens = textacy.extract.words(doc,

filter\_stops = True, # default True, no stopwords

filter\_punct = True, # default True, no punctuation

filter\_nums = True, # default False, no numbers

include\_pos = None, # default None = include all

exclude\_pos = None, # default None = exclude none

min\_freq = 1) # minimum frequency of words

output.append([str(word) for word in tokens]) # close output append

output[0:1]

* Create a frequency table of each of the tokens returned in this output. Below is some example code to get us started.

# all items

type(output)

# first list

type(output[0])

# first list, first item (this is the issue!)

type(output[0][0])

Counter(chain.from\_iterable(output))

**Summarize**

Write a paragraph explaining the process of cleaning data for an NLP pipeline. You should explain the errors you found in the dataset and how you fixed them. Explain the information that is gathered by using spacy and textacy and the final output. What did you learn from your frequency table? What is the text document about?

**Information Extraction**

**Libraries**

# libraries

import PyPDF2

import pandas as pd

import nltk

#nltk.download("punkt")

import re

import spacy

# only for datalore

import subprocess

#%%

print(subprocess.getoutput("python -m spacy download en\_core\_web\_sm"))

nlp = spacy.load("en\_core\_web\_sm")

import texacy

import summa

from summa import keywords

from snorkel.preprocess import preprocessor

from snorkel.types import DataPoint

from itertools import combinations

from snorkel.labeling import labeling\_function

from snorkel.labeling import PandasLFApplier

import networkx as nx

from matplotlib import pyplot as plt

**Import Text**

# creating a pdf file object

pdfFileObj = open('The\_Shadow\_Over\_Innsmouth.pdf', 'rb')

# creating a pdf reader object

pdfReader = PyPDF2.PdfReader(pdfFileObj)

# how many pages

len(pdfReader.pages)

# creating a page object

pageObj = pdfReader.pages

# extracting text from page

# loop here to get it all

text = []

for page in pageObj:

text.append(page.extract\_text())

# closing the pdf file object

pdfFileObj.close()

**Convert to Sentences and Pandas**

* ^ means start with
* [0-9] means any of these digits
* [a-zA-Z] means any alpha latin character lower or upper case
* $ ends with
* . mean any character
  + means zero or more of the previous character (so .\* means zero or more of any character)

# create a place to save the text

saved\_words = []

# loop over each word

for word in nltk.word\_tokenize(book):

# if the word starts with a number and ends with a letter

if (re.search(r'^[0-9].\*[a-zA-Z]$', word) != "None"):

# take out the numbers and save into our text

saved\_words.append(re.sub(r'[0-9]', '', word))

# if not then save just the word

else:

saved\_words.append(word)

book = ' '.join(saved\_words)

DF = pd.DataFrame(

nltk.sent\_tokenize(book),

columns = ["sentences"]

)

DF.head()

# for IE, we want sentence and/or paragraph level structure

**Part of Speech Tagging**

* Tag your data with spacy’s part of speech tagger.
* Convert this data into a Pandas DataFrame.

# easier to loop over the big text file than loop over words AND rows in pandas

spacy\_pos\_tagged = [(str(word), word.tag\_, word.pos\_) for word in nlp(book)]

# each row represents one token

DF\_POS = pd.DataFrame(

spacy\_pos\_tagged,

columns = ["token", "specific\_tag", "upos"]

)

* Use the dataframe to calculate the most common parts of speech.

DF\_POS['upos'].value\_counts()

* Use the dataframe to calculate if words are considered more than one part of speech (crosstabs or groupby).

DF\_POS2 = pd.crosstab(DF\_POS['token'], DF\_POS['upos'])

# convert to true false to add up how many times not zero

DF\_POS2['total'] = DF\_POS2.astype(bool).sum(axis=1)

#print out the rows that aren't 1

DF\_POS2[DF\_POS2['total'] > 1]

* What is the most common part of speech? ANSWER THIS IN YOUR TEXT
* Do you see words that are multiple parts of speech? ANSWER THIS IN YOUR TEXT

**KPE**

* Use textacy to find the key phrases in your text.
  + in the r window for r people
  + library(reticulate)
  + py\_install("networkx < 3.0", pip = T)

# textacy KPE

# build an english language for textacy pipe

en = textacy.load\_spacy\_lang("en\_core\_web\_sm", disable=("parser"))

# build a processor for textacy using spacy and process text

doc = textacy.make\_spacy\_doc(book, lang = en)

# text rank algorithm

print([kps for kps, weights in textacy.extract.keyterms.textrank(doc, normalize = "lemma", topn = 5)])

terms = set([term for term, weight in textacy.extract.keyterms.textrank(doc)])

print(textacy.extract.utils.aggregate\_term\_variants(terms))

* Use summa to find the key phrases in your text.

#TR\_keywords = keywords.keywords(book, scores = True)

#print(TR\_keywords[0:10])

* What differences do you see in their outputs? COMMENT ON HOW SLOW!
* Using textacy utilities, combine like key phrases. SEE ABOVE
* Do the outputs make sense given your text? ANSWER THIS QUESTION

**NER + Snorkel**

* Use spacy to extract named entities.
* Create a summary of your named entities.

# easier to loop over the big text file than loop over words AND rows in pandas

spacy\_ner\_tagged = [(str(word.text), word.label\_) for word in nlp(book).ents]

# each row represents one token

DF\_NER = pd.DataFrame(

spacy\_ner\_tagged,

columns = ["token", "entity"]

)

print(DF\_NER['entity'].value\_counts())

DF\_NER2 = pd.crosstab(DF\_NER['token'], DF\_NER['entity'])

print(DF\_NER2)

# convert to true false to add up how many times not zero

DF\_NER2['total'] = DF\_NER2.astype(bool).sum(axis=1)

#print out the rows that aren't 1

DF\_NER2[DF\_NER2['total'] > 1]

* Apply Snorkel to your data to show any relationship between names.

**get the data into a good format**

stored\_entities = []

# first get the entities, must be two for relationship matches

def get\_entities(x):

"""

Grabs the names using spacy's entity labeler

"""

# get all the entities in this row

processed = nlp(x)

# get the tokens for each sentence

tokens = [word.text for word in processed]

# get all the entities - notice this is only for persons

temp = [(str(ent), ent.label\_) for ent in processed.ents if ent.label\_ != ""]

# only move on if this row has at least two

if len(temp) > 1:

# finds all the combinations of pairs

temp2 = list(combinations(temp, 2))

# for each pair combination

for (person1, person2) in temp2:

# find the names in the person 1

person1\_words = [word.text for word in nlp(person1[0])]

# find the token numbers for person 1

person1\_ids = [i for i, val in enumerate(tokens) if val in person1\_words]

# output in (start, stop) token tuple format

if len(person1\_words) > 1:

person1\_ids2 = tuple(idx for idx in person1\_ids[0:2])

else:

id\_1 = [idx for idx in person1\_ids]

person1\_ids2 = (id\_1[0], id\_1[0])

# do the same thing with person 2

person2\_words = [word.text for word in nlp(person2[0])]

person2\_ids = [i for i, val in enumerate(tokens) if val in person2\_words[0:2]]

if len(person2\_words) > 1:

person2\_ids2 = tuple(idx for idx in person2\_ids)

else:

id\_2 = [idx for idx in person2\_ids[0:2]]

person2\_ids2 = (id\_2[0], id\_2[0])

# store all this in a list

stored\_entities.append(

[x, # original text

tokens, # tokens

person1[0], # person 1 name

person2[0], # person 2 name

person1\_ids2, # person 1 id token tuple

person2\_ids2 # person 2 id token tuple

])

DF['sentences'].apply(get\_entities)

# create dataframe in snorkel structure

DF\_dev = pd.DataFrame(stored\_entities, columns = ["sentence", "tokens", "person1", "person2", "person1\_word\_idx", "person2\_word\_idx"])

**figure out where to look (between and to the left)**

# live locate home road roads in at street (locations tied together)

# family terms for people

# get words between the data points

@preprocessor()

def get\_text\_between(cand: DataPoint) -> DataPoint:

"""

Returns the text between the two person mentions in the sentence

"""

start = cand.person1\_word\_idx[1] + 1

end = cand.person2\_word\_idx[0]

cand.between\_tokens = cand.tokens[start:end]

return cand

# get words next to the data points

@preprocessor()

def get\_left\_tokens(cand: DataPoint) -> DataPoint:

"""

Returns tokens in the length 3 window to the left of the person mentions

"""

# TODO: need to pass window as input params

window = 5

end = cand.person1\_word\_idx[0]

cand.person1\_left\_tokens = cand.tokens[0:end][-1 - window : -1]

end = cand.person2\_word\_idx[0]

cand.person2\_left\_tokens = cand.tokens[0:end][-1 - window : -1]

return cand

**figure out what to look for**

# live locate home road roads in at street (locations tied together)

# family terms for people

found\_location = 1

found\_family = -1

ABSTAIN = 0

location = {"live", "living", "locate", "located", "home", "road", "roads", "street", "streets", "in", "at", "of"}

@labeling\_function(resources=dict(location=location), pre=[get\_text\_between])

def between\_location(x, location):

return found\_location if len(location.intersection(set(x.between\_tokens))) > 0 else ABSTAIN

@labeling\_function(resources=dict(location=location), pre=[get\_left\_tokens])

def left\_location(x, location):

if len(set(location).intersection(set(x.person1\_left\_tokens))) > 0:

return found\_location

elif len(set(location).intersection(set(x.person2\_left\_tokens))) > 0:

return found\_location

else:

return ABSTAIN

family = {"spouse", "wife", "husband", "ex-wife", "ex-husband", "marry",

"married", "father", "mother", "sister", "brother", "son", "daughter",

"grandfather", "grandmother", "uncle", "aunt", "cousin",

"boyfriend", "girlfriend"}

@labeling\_function(resources=dict(family=family), pre=[get\_text\_between])

def between\_family(x, family):

return found\_family if len(family.intersection(set(x.between\_tokens))) > 0 else ABSTAIN

@labeling\_function(resources=dict(family=family), pre=[get\_left\_tokens])

def left\_family(x, family):

if len(set(family).intersection(set(x.person1\_left\_tokens))) > 0:

return found\_family

elif len(set(family).intersection(set(x.person2\_left\_tokens))) > 0:

return found\_family

else:

return ABSTAIN

# create a list of functions to run

lfs = [

between\_location,

left\_location,

between\_family,

left\_family

]

# build the applier function

applier = PandasLFApplier(lfs)

# run it on the dataset

L\_dev = applier.apply(DF\_dev)

L\_dev

DF\_combined = pd.concat([DF\_dev, pd.DataFrame(L\_dev, columns = ["location1", "location2", "family1", "family2"])], axis = 1)

DF\_combined

DF\_combined['location\_yes'] = DF\_combined['location1'] + DF\_combined["location2"]

DF\_combined['family\_yes'] = DF\_combined['family1'] + DF\_combined["family2"]

print(DF\_combined['location\_yes'].value\_counts())

print(DF\_combined['family\_yes'].value\_counts())

* What might you do to improve the default NER extraction?

**Knowledge Graphs**

**Slides Version**

* Based on the chosen text, add entities to a default spacy model.
* Add a norm\_entity, merge\_entity, and init\_coref pipelines.
* Update and add the alias lookup if necessary for the data.
* Add the name resolver pipeline.

**Or Use Your Snorkel Output**

* Create a co-occurrence graph of the entities linked together in your text.

# locations only

DF\_loc = DF\_combined[DF\_combined['location\_yes'] > 0]

DF\_loc = DF\_loc[['person1', 'person2']].reset\_index(drop = True)

cooc\_loc = DF\_loc.groupby(by=["person1", "person2"], as\_index=False).size()

# family only

DF\_fam = DF\_combined[DF\_combined['family\_yes'] > 0]

DF\_fam = DF\_fam[['person1', 'person2']].reset\_index(drop = True)

cooc\_fam = DF\_fam.groupby(by=["person1", "person2"], as\_index=False).size()

# take out issues where entity 1 == entity 2

cooc\_loc = cooc\_loc[cooc\_loc['person1'] != cooc\_loc['person2']]

cooc\_fam = cooc\_fam[cooc\_fam['person1'] != cooc\_fam['person2']]

print(cooc\_loc.head())

print(cooc\_fam.head())

* This creates a dataframe of node 1 and then node 2 (entity 1 to entity 2) and then frequency (size)

# start by plotting the whole thing for location

cooc\_loc\_small = cooc\_loc[cooc\_loc['size']>1]

graph = nx.from\_pandas\_edgelist(

cooc\_loc\_small[['person1', 'person2', 'size']] \

.rename(columns={'size': 'weight'}),

source='person1', target='person2', edge\_attr=True)

pos = nx.kamada\_kawai\_layout(graph, weight='weight')

\_ = plt.figure(figsize=(20, 20))

nx.draw(graph, pos,

node\_size=1000,

node\_color='skyblue',

alpha=0.8,

with\_labels = True)

plt.title('Graph Visualization', size=15)

for (node1,node2,data) in graph.edges(data=True):

width = data['weight']

\_ = nx.draw\_networkx\_edges(graph,pos,

edgelist=[(node1, node2)],

width=width,

edge\_color='#505050',

alpha=0.5)

plt.show()

plt.close()

# start by plotting the whole thing for location

graph = nx.from\_pandas\_edgelist(

cooc\_fam[['person1', 'person2', 'size']] \

.rename(columns={'size': 'weight'}),

source='person1', target='person2', edge\_attr=True)

pos = nx.kamada\_kawai\_layout(graph, weight='weight')

\_ = plt.figure(figsize=(20, 20))

nx.draw(graph, pos,

node\_size=1000,

node\_color='skyblue',

alpha=0.8,

with\_labels = True)

plt.title('Graph Visualization', size=15)

for (node1,node2,data) in graph.edges(data=True):

width = data['weight']

\_ = nx.draw\_networkx\_edges(graph,pos,

edgelist=[(node1, node2)],

width=width,

edge\_color='#505050',

alpha=0.5)

plt.show()

plt.close()

**Text Summarization**

**Find Text**

import pysrt

import pandas as pd

import re

from sentence\_transformers import SentenceTransformer

# install faiss-cpu

import faiss

import time

subs = pysrt.open("bodies.srt")

DF = pd.DataFrame([

{

"Text": sub.text

} for sub in subs])

DF

def remove\_noise(text):

text = re.sub("<.\*>", " ", text)

text = re.sub("{.\*}", " ", text)

text = re.sub("\[.\*\]", " ", text)

text = text.strip()

return text

DF['clean'] = DF['Text'].apply(remove\_noise)

DF = DF[DF['clean'] != ""]

DF

Create A Search Engine

Using each sentence as your “documents”, create a search engine to find specific pieces of text.

# this is creating the embeddings

model = SentenceTransformer('msmarco-MiniLM-L-12-v3')

bodies\_text\_embds = model.encode(DF['clean'].to\_list())

# Create an index using FAISS

index = faiss.IndexFlatL2(bodies\_text\_embds.shape[1])

index.add(bodies\_text\_embds)

faiss.write\_index(index, 'index\_bodies')

bodies\_text\_embds

# define a search

def search(query, k):

t=time.time()

query\_vector = model.encode([query])

top\_k = index.search(query\_vector, k)

print('totaltime: {}'.format(time.time()-t))

return [DF['clean'].to\_list()[\_id] for \_id in top\_k[1].tolist()[0]]

Search for several items.

search("cop", 10)

search("gun", 10)

search("car", 10)

Examine the results and comment on how well you think the search engine worked.

ANSWER THIS QUESTION