Welcome to The Carpentries Etherpad!

This pad is synchronized as you type, so that everyone viewing this page sees the same text. This allows you to collaborate seamlessly on documents.

Use of this service is restricted to members of The Carpentries community; this is not for general purpose use (for that, try https://etherpad.wikimedia.org).

Users are expected to follow our code of conduct: https://docs.carpentries.org/topic_folders/policies/code-of-conduct.html

---

Welcome to the workshop!

Links

- Lesson materials: https://carpentries-incubator.github.io/python-text-analysis/
- Workshop Setup: https://carpentries-incubator.github.io/python-text-analysis/setup.html
- Intro Slides: https://docs.google.com/presentation/d/1Wb_pv5ju0dKhLqtz3_dipNFTfAoeaSU8EbAcFWI7jlE/edit?usp=sharing
- Corpus Development Slides: https://docs.google.com/presentation/d/1qwYJ7j11bkCKoBS5Db6VMslziasR_H2wKIN9ny8b7rI/edit?usp=sharing
- Daily feedback: https://forms.gle/zdjBKka8tyRBHeV88
- TF-IDF, LSA, Word2Vec Slides: https://docs.google.com/presentation/d/1kpWnIpIsMdm_7slOrfUAqL9qtoSAF815xd4a7zftahk/edit?usp=sharing
- Post-workshop slides: https://docs.google.com/presentation/d/1N5-VoxTKnSqJModxwr1W3AtepnwdTpR1rq4y95CspMs/edit?usp=sharing

Day 1

Sign in

Name, department/program/affiliation, describe your work in 1-2 sentences

- Carleigh Young, Ed Psych LAMP MS student, also admissions tech specialist for Fairmont State University. I do systems and process improvements in our enrollment management systems to

improve enrollment strategies along with descriptive/predictive modeling for enrollment
- Jennifer Patiño, UW-Madison Libraries, Data and Digital Scholarship Librarian, helping people with their research data, including help finding data for their projects
- Nikki Garlic, Judicial Studies, Teaching Assistant Professor, University of Nevada, Reno. I study equal access to the court system using computational text analysis, and teach research methods.
- Ann Hanlon, UWM Libraries, Digital Collections and DH. I work with archival collections and digitization as well as use of those collections, digital preservation, and scholarship.
- Chris Endemann, Data Science Facilitator @ Data Science Hub, I help researchers implement data science and machine tools for their research.
- Karl Holten, UWM Libraries & Letters and Science. Systems administrator with MS in Computer Science, specializing in Natural Language Processing.
- Zekai Otles-UW-Madison DOIT Research Cyberinfrastructue consultant
- Jennifer Scheuren Midwest Center for Cryo Electron Tomography HPC administrator
- Sam Nycklemoe, Biomedical Data Science Masters Student. Working on research with clinical trial design, moving to text embedding in clinical setting during the summer.
- Ann Wieben, Nursing, Post Doc, intersection of AI and nursing practice
- Mengyu Li, a Ph.D. student in the School of Journalism and Mass Communication. Her research interests include computational communication, communication technology, and media psychology.
- Byron Crites, Data Scientist, UW School of Medicine, parsing medical notes.
- Mary Roth, UW-Madision Dept of Anesthesiology, Research Administrator/Data Science Facilitator
- Chaitanyasuma Jain, MS student, UW Madison
- Russell Dimond, Statistical Consultant in the Social Science Computing Cooperative at UW-Madison. Supporting researchers as they do text analysis.
- Grace Cagle, Post Doc, Dept. of Soil Science at UW-Madison, microbial ecology & bioinformatics
- Michael Place, Great Lakes Bioenery Research Center at UW-Madison
- Scott Prater, UW Madison Libraries, UW Digital Collections Center. I work on strategy, project management, and implementation of digital library tools and infrastructure.
- Lydia Reimer, Entrepreneurship Science Lab. Analyzing student essays

Notes

Before we get started, we would like to also provide a disclaimer. The humanities involves a wide variety of fields. Each of those fields brings a variety of research interests and methods to focus on a wide variety of questions. AI is not infallible or without bias. NLP is simply another tool you can use to analyze texts and should be critically considered in the same way any other tool would be. The goal of this workshop is not to replace or discredit existing humanist methods, but to help humanists learn new tools to help them accomplish their research.

Interpretive loop: https://carpentries-incubator.github.io/python-text-analysis/01-basicConcepts/index.html

- Variety of tasks possible with NLP
Task -> Collect Relevant Data -> Preprocess Text -> Embed Text -> NLP Task with Embedding -> Results -> Interpretration

HuggingFace is a very popular wesbite for sharing machine learning models and datasets. The go-to option for NLP-related models.

Open a Colab notebook...

```
from transformers import pipeline, Conversation
converse = pipeline("conversational", model="microsoft/DialoGPT-medium")

conversation1 = Conversation("Going to the movies tonight- any suggestions?")
conversation2 = Conversation("What's the last book you have read?")

converse([conversation1 , conversation2])
```

What tasks can NLP do?
- Search
- Topic modeling
- Token classification
- Document summarization
- Text prediction

Group Activity and Discussion

With some experience with a task, let's get a broader overview of the types of tasks we can do. Relaunch a web browser and go back to https://huggingface.co/tasks. Look at a couple of tasks for HuggingFace. The groups will be based on general categories for each task. Discuss possible applications of this type of model to your field of research. Try to brainstorm possible applications for now, don't worry about technical implementation.

1. Tasks that seek to convert non-text into text
    - https://huggingface.co/tasks/image-to-text
    - https://huggingface.co/tasks/text-to-image
    - https://huggingface.co/tasks/automatic-speech-recognition
    - https://huggingface.co/tasks/image-to-text
2. Searching and classifying documents as a whole
    - https://huggingface.co/tasks/text-classification
    - https://huggingface.co/tasks/sentence-similarity
3. Classifying individual words- Sequence based tasks
    - https://huggingface.co/tasks/token-classification
    - https://huggingface.co/tasks/translation
4. Interactive and generative tasks such as conversation and question answering
    - https://huggingface.co/tasks/text-generation
    - https://huggingface.co/tasks/question-answering

Briefly present a summary of some of the tasks you explored. What types of applications could you see this type of task used in? How might this be relevant to a research question you have? Summarize these tasks and present your findings to the group.

Building a Corpus

- The best sources to build a corpus, or dataset, for text analysis will ultimately depend on the needs of your project. Datasets and sources are not usually prepared to be used in specific kinds of projects, therefore the burden is on the researcher to select materials that will be suitable for their corpus.

- Avoid the temptation to grab data in bulk w/out evaluating it. It's important to think critically about the data you are gathering, its context, and whether or not it is relevant

- eval content, file type, bias, rights and permissions, quality, and features

- your research question will inform the content of your corpus and potential data sources

- Ex: "How are women represented in 10th century documents?" vs "How are women represented in classic 10th century American novels?" Narrowing the scope can help you build a more reasonably sized corpus without spending forever on the data collection process

Are there tips for searching for bias in datasets?
- It can be difficult to know without first exploring the data. This is why we sometimes call it an "analysis loop". Bias can be evaluated in many ways. One method is to use a text embedding method like Word2Vec to see the kind of meaning represented by sensitive words/topics.
- things to pay attention to include the time frame for the data set, limitations in terms of how it was collected (for example, census questions changed from census to census and data is limited for the first 75 years), who collected it, etc. So its provenance, really. If you have access to that information.

Do you have the rights to use the data?
- Building Legal Literacies for Text Data Mining: https://berkeley.pressbooks.pub/buildinglltdm/
- https://buildinglltdm.org/
- Consider copyright, licensing, terms of use, tecnology protection measures, privacy, and research protections

Assessing Bias
- who created the data and why? what biases might they hold?
- any bias you might introduce when assembling the corpus
- when using pretrained models, consider biases in datasets used to train models. If there are known issues, consider options for remediation or look for alternative models

Data Quality and Features
- Ebook from "Emma" by Jane Austen vs OCR (optimal character recognition) from a digitized newspaper article
- https://www.gutenberg.org/cache/epub/158/pg158.txt (Ebook): standardized text (page numbers removed and some other standardization procedures; good quality in general)

Preparing and Preprocessing Data
- open a colab notebook from google drive (within your text-analysis folder)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
from os import listdir
wksp_dir = '/content/drive/My Drive/Colab Notebooks/text-analysis/code'
listdir(wksp_dir)

import sys
sys.path.insert(0, wksp_dir)

!pip install pathlib parse # parse is used by one of the helper functions

import glob
import os
from pathlib import Path

from helpers import create_file_list

corpus_dir = '/content/drive/My Drive/Colab Notebooks/text-analysis/data/books'
corpus_file_list = create_file_list(corpus_dir)
print(corpus_file_list)

austen_list = create_file_list(corpus_dir, 'austen*')
print(austen_list)

preview_len = 290
emmapath = create_file_list(corpus_dir, 'austen-emma*')[0]
print(emmapath)

sentence = ""
with open(emmapath, 'r') as f:
```

- `sentence = f.read(preview_len)[50:preview_len]`

```
print(sentence)
```

Preprocess steps
1. Tokenization: break text into words or "tokens"
2. Stems and lemmas: Reduce common variations of words to their root words without any conjugation
3. Remove stop words ("a", "the", etc.) and punctuation

```
import spacy
import en_core_web_sm
spacyt = spacy.load("en_core_web_sm")

class Our_Tokenizer:
  def _init_(self):
    self.nlp = en_core_web_sm.load()
    self.nlp.max_length = 4500000
  def _call_(self, document):
    tokens = self.nlp(document)
    return tokens
```

```
# using spacy's tokenizer function
tokens = spacyt(sentence)
for t in tokens:
```

- print(t.text)

```
for t in tokens:
  print(t.lemma) # prints token ID number

for t in tokens:
  print(t.lemma_) # prints value/lemma stored rather than ID number

# Removing stop words
from spacy.lang.en.stop_words import STOP_WORDS
print(STOP_WORDS)

# maybe we want to add stop words to this list. Let's try adding "zebra" to the list of words we want to exclude from our analysis

z= spacyt("zebra")[0]
print(z.is_stop) # false

STOP_WORDS.add("zebra")
spacyt = spacy.load("en_core_web_sm") # refresh model
z = spacyt("zebra")[0]
print(z.is_stop) # true

# remove zebra from stop words
STOP_WORDS.remove("zebra")
spacyt = spacy.load("en_core_web_sm") # refresh model
z = spacyt("zebra")[0]
print(z.is_stop) # true

# removing proper nouns that do not tell us anything interesting
STOP_WORDS.add("emma")
spacyt = spacy.load("en_core_web_sm")

# retokenize our sentence
tokens = spacyt(sentence)

for token in tokens:
```

- if not token.is_stop and not token.is_punct:
    - print(str.lower(token.lemma_))

Parts of Speech

While we can manually add Emma to our stopword list, it may occur to you that novels are filled with

characters with unique and unpredictable names. We've already missed the word "Woodhouse" from our list. Creating an enumerated list of all of the possible character names seems impossible. One way we might address this problem is by using Parts of speech (POS) tagging. POS are things such as nouns, verbs, and adjectives. POS tags often prove useful, so some tokenizers also have built in POS tagging done. Spacy is one such library. These tags are not 100% accurate, but they are a great place to start. Spacy's POS tags can be used by accessing the pos_ method for each token.

for token in tokens:

- if token.is_stop == False and token.is_punct == False:
    - print(str.lower(token.lemma_) + " " + token.pos_)

class Our_Tokenizer:

- def __init__(self): # import spacy tokenizer/language model
    - self.nlp = en_core_web_sm.load()
    - self.nlp.max_length = 4500000 # increase max number of characters that spacy can process (default = 1,000,000)
- def __call__(self, document):
    - tokens = self.nlp(document)
    - simplified_tokens = [
        - #our helper function expects spacy tokens. It will take care of making them lowercase lemmas. token for token in tokens
        - if not token.is_stop
        - and not token.is_punct
        - and token.pos_ in {"ADJ", "ADV", "INTJ", "NOUN", "VERB"}
    -]
    - return simplified_tokens

```
tokenizer = Our_Tokenizer()
tokens = tokenizer(sentence)
print(tokens)
```

```
from helpers import parse_into_dataframe
pattern = "/content/drive/My Drive/Colab Notebooks/text-analysis/data/books/{author}-{title}.txt"
data = parse_into_dataframe(pattern, corpus_file_list)
data["Lemma_File"] = lemma_file_list
```

Vector Space

```
import numpy as np
import matplotlib.pyplot as plt
corpus = np.array([[1,10],[8,8],[2,2],[2,2]])
print(corpus)
```

```
# matplotlib expects a list of values by column, not by row.
# We can simply turn our table on its edge so rows become columns and vice versa.
corpusT = np.transpose(corpus)
print(corpusT)
```

```
X = corpusT[0]
Y = corpusT[1]

# define some colors for each point. Since points A and B are the same, we'll have them as the same color.mycolors = ['r','g','b','b']

# display our visualization
plt.scatter(X,Y, c=mycolors)
plt.xlim(0, 12)
plt.ylim(0, 12)
plt.show()
```

TF-IDF

The method of using word counts is just one way we might embed a document in vector space. Let's talk about more complex and representational ways of constructing document embeddings. To start, imagine we want to represent each word in our model individually, instead of considering an entire document. How individual words are represented in vector space is something called "word embeddings" and they are an important concept in NLP.

Currently our model assumes all words are created equal and are all equally important. However, in the real world we know that certain words are more important than others.
For example, in a set of novels, knowing one novel contains the word *the* 100 times does not tell us much about it. However, if the novel contains a rarer word such as *whale* 100 times, that may tell us quite a bit about its content.
A more accurate model would weigh these rarer words more heavily, and more common words less heavily, so that their relative importance is part of our model.
However, rare is a relative term. In a corpus of documents about blue whales, the term *whale* may be present in nearly every document. In that case, other words may be rarer and more informative. How do we determine how these weights are done?

One method for constructing more advanced word embeddings is a model called TF-IDF.

TF-IDF stands for term frequency-inverse document frequency. The model consists of two parts: term frequency and inverse document frequency. We multiply the two terms to get the TF-IDF value.

Term frequency is a measure how frequently a term occurs in a document. The simplest way to calculate term frequency is by simply adding up the number of times a term occurs in a document, and dividing by the total word count in the corpus.

Inverse document frequency measures a term's importance. Document frequency is the number of documents a term occurs in, so inverse document frequency gives higher scores to words that occur in fewer documents. This is represented by the equation:
IDF(x) = ln[(N+1) / (DF(x)+1)]
where…

- N represents the total number of documents in the corpus
- DF(x) represents document frequency for a particular term/word, x. This is the number of

documents a term occurs in.

The key thing to understand is that words that occur in many documents produce smaller IDF values since the denominator grows with DF(x).

We can also embed documents in vector space using TF-IDF scores rather than simple word counts. This also weakens the impact of stop-words, since due to their common nature, they have very low scores. Now that we've seen how TF-IDF works, let's put it into practice.

```
from pandas import read_csv
data = read_csv("/content/drive/My Drive/Colab Notebooks/text-analysis/data/data.csv")
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(input='filename', max_df=.6, min_df=.1) # initialize TF-IDF vectorizer.
```

Here, max_df=.6 removes terms that appear in more than 60% of our documents (overly common words like the, a, an) and min_df=.1 removes terms that appear in less than 10% of our documents (overly rare words like specific character names, typos, or punctuation the tokenizer doesn't understand). We're looking for that sweet spot where terms are frequent enough for us to build theoretical understanding of what they mean for our corpus, but not so frequent that they can't help us tell our documents apart.

```
tfidf = vectorizer.fit_transform(list(data["Lemma_File"]))
print(tfidf.shape) # Here, tfidf.shape shows us the number of rows (books) and columns (words) are in our model.
```

```
vectorizer.get_feature_names_out()[0:5] # print first 5 words included in TF-IDF table
```

```
print(vectorizer.idf_[0:5]) # weights for each token
```

```
from pandas import DataFrame
tfidf_data = DataFrame(vectorizer.idf_, index=vectorizer.get_feature_names_out(), columns=["Weight"])
tfidf_data
```

Day 2

Sign in (Name, affiliation, and any questions about yesterday's material)

- Chris Endemann (he/him), Data Science Facilitator @ Data Science Hub, UW-Madison
- Karl Holten (he/him), Systems Administrator, UWM Libraries and L&S
- Ann Hanlon, UWM Libraries
- Russell Dimond (he/him), SSCC
- Trisha Adamus (she/her), Ebling Library, UW-Madison
- Grace Cagle (She/her), UWM Soil Science, tokenizing non-language text
- Jennifer Scheuren

- Todd Hayes, SMPH
- Lydia Reimer, Entrepreneurship Science Lab
- Ann Wieben, Nursing, no specific questions at this time
- Sam Nycklemoe, BDS
- Jean-Yves Sgro, (observer) Biochemistry Dept & Biotechnology Center
- Scott Prater, UW Digital Collections Center
- Jennifer Patiño, (She/her), Data and Digital Scholarship Librarian, UW-Madison Libraries
- Mengyu Li (she/her)
- Nikki Garlic (she/her), tokenizing for multiple languages in the same project
- Zekai Otles (He/him) -UW-Madison DoIT Research Cyberinfrastructure
- Todd Hayes

Notes

Yesterday we discussed preprocessing and several embeddings type.

We learned word counts and TF-IDF. TF-IDF words with high scores are frequent in a document and rare across the corpus. Emphasizes words that are more meaningful and unique. A low TF-IDF score means a term that is common across many documents and not unique.

TF-IDF is foundational for our next section on LSA - Latent Semantic Analysis. TF-IDF scores create a matrix, but that can be difficult to summarize. Next, we'll be look at options to help gain insights, including topic modeling

Features: synonymous with dimensions or columns. Attributes of the text that we can represent numerically. Features = dimensions = columns.

We've covered word counts; weighted word counts (TF-IDF - looking at uniqueness). So far, we've just gotten to the text embedding procesess in our process loop.

Now to topic modeling:

There are several variants of topic modeling - we'll look at LSA today. Distills a TF-IDF matrix down to a smaller handful of "latent" or hidden topics present in a corpus. TF-IDF is typically the input for the LSA algorithm.

Sparse vectors: not every word appears in every document
Dense: Every word has a score assigned to it and a topic score.

We need to decide if these representations are meaningful. Word presence only doesn't give enough context; importance of terms gets us a little further; semantic topics is more meaningful for describing what documents are about. Still limited because none of these methods so far consider word order. (Word2Vec, coming up! will do that)

Applying "dimensionality reduction" to a large matrix. This means sacrificing some information, and in the map example, different projections will favor or lose some information. But goal is to preserve

relevant information and to make sense of the object.

Create a new colab notebook.
Copy and paste the code below

```
# Run this cell to mount your Google Drive.
from google.colab import drive
drive.mount('/content/drive')

# Show existing colab notebooks and helpers.py file
from os import listdir
wksp_dir = '/content/drive/My Drive/Colab Notebooks/text-analysis/code'
print(listdir(wksp_dir))

# Add folder to colab's path so we can import the helper functions
import sys
sys.path.insert(0, wksp_dir)

# Read the data back in.
from pandas import read_csv
data = read_csv("/content/drive/My Drive/Colab Notebooks/text-analysis/data/data.csv")
data.head()
```

next:

```
!pip install pathlib parse # parse is used by helper functions
```

First, we'll reproduce yesterday's TF-IDF matrix.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(input='filename', max_df=.6, min_df=.1)
```

- *# Here, max_df=.6 removes terms that appear in more than 60% of our documents (overly common words like the, a, an) and min_df=.1 removes terms that appear in less than 10% of our documents (overly rare words like specific character names, typos, or punctuation the tokenizer doesn't understand)*

```
tfidf = vectorizer.fit_transform(list(data["Lemma_File"]))
print(tfidf.shape)
```

#SVD is a linear algebra dimensional reduction technique. SVD is used to combine multiple columns into a #smaller set of columns that attempts to accurately capture the original relationships in the original matrix.

```
from sklearn.decomposition import TruncatedSVD
maxDimensions = tfidf.shape[0]-1

svdmodel = TruncatedSVD(n_components=maxDimensions, algorithm="arpack")
```

- #the arpack algorithm is very efficient when working with sparse and large matrices. TF-IDF will always be a sparse matrix.

```
lsa = svdmodel.fit_transform(tfidf)
print(lsa)
print (lsa.shape)
```

- #Now we have reshaped our data to be represented in 40 dimensions rather than 10,000!

```
import matplotlib.pyplot as plt
import numpy as np
# this shows us the amount of dropoff in explanation we have in our sigma matrix.
print(svdmodel.explained_variance_ratio_) # Calculate cumulative sum of explained variance ratio
cumulative_variance_ratio = np.cumsum(svdmodel.explained_variance_ratio_)
plt.plot(range(1, maxDimensions + 1), cumulative_variance_ratio * 100)
plt.xlabel("Number of Topics")
plt.ylabel("Cumulative % of Information Retained")
plt.ylim(0, 100)  # Adjust y-axis limit to 0-100
plt.grid(True)    # Add grid lines
```

- #Trying to calculate the cumulative sum of the variable to plot it. We have forty new dimensions, if we keep them all we will have 100% of our data. Want something more like 5 or 10 topics to describe our text. Want to have enough information in your dataset, but simplified to understand it. The dimensions are arranged in descending order of how much of our corpus they explain. We can pick how many topics we want by looking for the "elbow" or inflection point - where the slope decreases significantly, and choose the number of dimensions there. For our chart, for example, we'll keep 7 total. We are tossing out information! We are not considering the full spectrum, but considering a significant amount. Looking to summarize and see general patterns, not describe the whole dataset.

```
numDimensions = 7
svdmodel = TruncatedSVD(n_components=numDimensions, algorithm="arpack")
lsa = svdmodel.fit_transform(tfidf)
print(lsa)
```

- We have seven new dimensions - and we'll refer to them as **topics**.

- We'll add these seven new dimensions to our data frame.

```
data[["X", "Y", "Z", "W", "P", "Q"]] = lsa[:, [1, 2, 3, 4, 5, 6]]
data.head()
```

- (*lesson update note: change max dimensions to "6" in lesson!)
- We are going to plot the data to "center" the data.

```
data[["X", "Y", "Z", "W", "P", "Q"]] = lsa[:, [1, 2, 3, 4, 5, 6]]-lsa[:, [1, 2, 3, 4, 5, 6]].mean(0)

data[["X", "Y", "Z", "W", "P", "Q"]].mean()

data.to_csv("/content/drive/My Drive/Colab Notebooks/text-analysis/data/data.csv",
```

index=False)

- Note: Different Google Colab hardware will produce slightly different means. Exact replication - if that's important think about using Docker.
- To plot our results:

```
from helpers import lsa_plot
lsa_plot(data, svdmodel)
```

- And we can help unpack the results a bit more by assigning a unique color to each of our authors:

```
colormap = {
   "austen": "red",
   "chesterton": "blue",
   "dickens": "green",
   "dumas": "orange",
   "melville": "cyan",
   "shakespeare": "magenta"
}

lsa_plot(data, svdmodel, groupby="author", colors=colormap)
```

To see the topics that are strongest here, we can use a helper functions called showtopics - this is what it looks like:

```
import pandas as pd
def show_topics(vectorizer, svdmodel, topic_number, n):
   # Get the feature names (terms) from the TF-IDF vectorizer    terms =
vectorizer.get_feature_names_out()

   # Get the weights of the terms for the specified topic from the SVD model
```

- weights = svdmodel.components_[topic_number]

```
   # Create a DataFrame with terms and their corresponding weights
```

- df = pd.DataFrame({"Term": terms, "Weight": weights})

```
   # Sort the DataFrame by weights in descending order to get top n terms (largest
positive weights)
```

- highs = df.sort_values(by=["Weight"], ascending=False)[0:n]

```
   # Sort the DataFrame by weights in ascending order to get bottom n terms
(largest negative weights)
```

- lows = df.sort_values(by=["Weight"], ascending=False)[-n:]

```
   # Concatenate top and bottom terms into a single DataFrame and return
```

- **return** pd.concat([highs, lows])

 # Get the top 5 and bottom 5 terms for each specified

- *topic*topic_words_x = show_topics(vectorizer, svdmodel, 1, 5) *# Topic 1*
- topic_words_y = show_topics(vectorizer, svdmodel, 2, 5) *# Topic 2*

We'll use:

from helpers import show_topics
topic_words_x = show_topics(vectorizer, svdmodel, topic_number=1, n=5)
topic_words_y = show_topics(vectorizer, svdmodel, topic_number=2, n=5)

print(topic_words_x)

^note: this feature returns none and just prints things

We can see how a certain amount of "human in the loop" is necessary to really unpack the dimensions. For our scatterplot, we can interpret the outlying Shakespeare dots as representing Elizabethan language versus 19th century English.

The results might also prompt us to go back and do additional pre-processing. Return to the interpretive loop! To help explain why you're seeing what you're seeing. You can also try increasing the number of words (change n=10), for example.

note: Questions re: short documents and the utility of LSA. (to think about for the workshop) And we'll also look at other text analysis methods that might be more applicable as we proceed. Add some text re: differences between LDA and LSA.

Word2Vec

New approach! Chris is sharing a pre-filled Colab notebook.
https://colab.research.google.com/drive/17eTHd9IZazoLCEsn5Gz_GmVH4L0BLSly?usp=sharing

Click on Copy to Drive in order to use (to the right of Code and Text)

Bag-of-words limitations

so far, we aren't considering the order of how words appear. Only weighting and frequency. But to understand meaning, it makes sense to consider the context that words are appearing in. We speak in sentences, after all.

Word2Vec is inspired by the "distributional hypothesis". Introduced in 2013. A good option if you'r lower on compute than what is required for LLMs. it is focused on feature representation per word.

Enter the neural network! There is a supplemental episode that goes more deeply into how neural networks work. Today we're going to learn to drive, but not how the engine works.

NN's learn functions that can map a set of input features to some output. This gives them a "general capability." Prediction and classification. They learn new meaningful features from the input data.

Task, for example: Given the surrounding words, predict the center word; given the center word, predict the surrounding words. We are using the learned features as our embedding.

The vectors learned by these models are a reflection on the specific data the model was trained on. These learned features, or vectors, are considered black boxes and are very abstract, lack direct interpretability.

For our tasks:
Analysis relevance - since we're using this tool, your analysis should focus on specific words.
Data quality - ensure high quality data; garbage in/garbage out in this case! Noisy data will not get a good result
Corpus size: Performs better with larger corpora, improves the quality of learned word vectors.
Domain specificity: Choose a dataset relevant to your research.

https://carpentries-incubator.github.io/python-text-analysis/07-wordEmbed_intro/index.html (see links embedded in Word2Vec Applications exercise)

- **Semantic Change Over Time**: How have the meanings of words evolved over different historical periods? By training Word2Vec models on texts from different time periods, researchers can analyze how word embeddings change over time, revealing shifts in semantic usage.
- Authorship Attribution: Can Word2Vec be used to identify the authors of anonymous texts or disputed authorship works? By comparing the word embeddings of known authors' works with unknown texts, researchers can potentially attribute authorship based on stylistic similarities (e.g., Agrawal et al., 2023 and Liu, 2017).
- **Authorship Attribution**: Word2Vec has been applied to authorship attribution tasks (e.g., Tripto and Ali, 2023).
- **Comparative Analysis of Multilingual Texts**: Word2Vec enables cross-lingual comparisons. Researchers have explored multilingual embeddings to study semantic differences between languages (e.g., Heijden et al., 2019).
- **Studying Cultural Concepts and Biases**: Word2Vec helps uncover cultural biases in language. Researchers have examined biases related to race, religion, and colonialism (e.g., Petreski and Hashim, 2022).

Using Gensim library; trained on a Google News dataset.

The vectors are abstract and we don't know how to interpret them. We can use cosine similiarity to see how words/vectors (?) compare.

90 degrees (orthogonal); very different from one another
180 degrees: anti each other :)

A higher score indicates higher similarity. Comparing on a relative scale.

Use a gensim function called "most similar" to find the top ten highest similarity terms associated with the vector - or whale.

This also helps to make sure that the vector really is meaningful (again - human in the loop to determine that, based on domain knowledge)

We can have vectors that have 300 features per word, but it's hard to visualize more than three dimensions. We'll use PCA (prinicipal component analysis) to represent a transformed version of that vector with fewer dimensions to help us make sense of it.

Training Word2Vec

New Notebook: https://colab.research.google.com/drive/1DYKs6-qSSdSZOLNdB83Ue1-4ltXZr6rS?usp=sharing

We'll train a word2vec model from scratch using Moby Dick
We;ll extract the row that contains Moby Dick from our original data set

Splitting text into sentences: NLTK is one library you can use, using the NTLK 'punkt' sentence tokenizer. You can check out the helpers.py file to see what kinds of pre-processing it will do.

type(sentences) - the type is a list. But you'll use a pandas function to convert that to series. So every sentence in the book Moby Dick will get pre-processed as a series.

CBOW: continuous bag of words
sk: Skip-gram (can work better with rare words)

Word2Vec isn't computationally intensive - can be run on a laptop, unlike some of the models we'll look at tomorrow.

Downsampling is a tradeoff, of course, but as long as you have a large enough data set it shouldn't hurt too much (always dependent on your task/research question)

Word2Vec will only produce vector representations for words used in the datat to train the model. So you can't study words that are not included in the corpus with this tool.

(A solution is to use fastText, another model from Gensim using n-grams (components of words))?

To check if a word is part of your corpus, try https://stackoverflow.com/questions/30301922/how-to-check-if-a-key-exists-in-a-word2vec-trained-model-or-not

To find a rare word, one tip from Jean-Yves:

FYI - Using ‘grep’ we can see that squid appears exactly in 4 lines… Also called “cuttle-fish” on that line:

whale--squid or cuttle-fish--lurks at the bottom of that sea, because
Here is the command:

!grep squid "drive/My Drive/Colab Notebooks/text-analysis/data/books/melville-moby_dick.txt"

Day 3

Sign in (Name, affiliation, and any questions about yesterday's material)

- Jean-Yves Sgro, Biochemistry / Biotechnology Center
- Chris Endemann, Data Science Hub
- Karl Holten, UW-Milwaukee, Library and L&S
- Ann Hanlon, UWM Libraries
- Jennifer Scheuren MCCET
- Jennifer Patiño, UW-Madison Libraries
- Ann Wieben, Nursing
- Grace Cagle, UW-Madison Soil Science
- Byron Crites, UW Madison
- Zekai Otles, UW-Madison, DOIT Research Cyberinfrastructure
- Mengyu Li, SJMC
- Nikki Garlic, UNR
- u
- Lydia Reimer, ESLAB

Notes

Link to BERT explainer:
https://colab.research.google.com/drive/1CH4z1zRWmw9I6sVZSRE3kgDYYYx6HT_8?usp=sharing

What is a large language model or LLM? It is a sophisticated neural network model with an achitecture that is known as a "transformer model". BERT is one kind of LLM model.

BERT stands for Bidirectional Encoder Representations from Transformers, is based on transformers, a deep learning model in which every output element is connected to every input element, and the weightings between them are dynamically calculated based upon their connection.

We'll use the Hugging Face platform to access LLMs. Specifically, we'll play around with the BERT model.

Before we process text through an LLM, we tokenize our text just as we did with previous methods.

After tokenizing, we can create an embedding for each token. We also embed segments or sequences of tokens to get greater context embedded to the model. FInally, we embed the position of each token so that we can understand how words come together in a sentence / in order.

*Does BERT run locally, or are API calls made to a remote instance? It runs locally as we are calling it in our colab notebook.

*I sometimes work with multi-lingual data. I'm wonder if there are LLMs designed to work across languages or if they are all language specific. Yes, this is possible. LLMs can make sense of any language provided you have enough training data. Here is a pretrained model that works with over 100 languages: https://huggingface.co/google-bert/bert-base-multilingual-cased

BERT finetuning:
https://colab.research.google.com/drive/1OmCy3FPUOZML2mB_nwh1ryHjCRzaQc6-?usp=sharing

GPU = graphical processing unit. They are amazingly fast at doing matrix computations which is essentially all we ever do with neural networks. Make sure to set the colab notebook to use the GPU so that you can run the code much faster.

Fine Tuning allows you to adapt tasks that aren't already present in the model.

nfo on B-I-O tags explained in: https://medium.com/analytics-vidhya/bio-tagged-text-to-original-text-99b05da6664

What is an epoch? An epoch is one we train our neural network on a single pass of our training dataset. When we train neural networks, we typically want to give he model multiple chances to see each example in our data. This leads to better performance.

Validation data is used to assess overfitting of neural networks as we are training them. Test data is used to get a final evaluation of a model's generalizeability.

If you encounter overfitting, you'll see that your validation error is very high. In such a scenario, you would want to decrease the number of epochs and retrain the model to assess the validation error again.

Advice for all: try not to get too bogged down by all of the parameter configurations of neural networks. There are many design choices, but the models can be quite robust and work with a variety of configurations. Getting neural networks to work is a very empirical science. You basically just need to try a handful of settings with your specific data and see what works best. Just don't go too crazy and try to optimize every single parameter -- it isn't really necessary in most cases.