Nina Austria
Abhishek Chakraborty
STAT 399
Winter 2024

Notes From DataCamp Courses: Winter 2024

Weeks 2-3: Got started with a course called "Understanding Artificial Intelligence" which is more of a theoretical course – talks about AI's challenges, societal implications, machine learning, deep learning, generative models and more

"Understanding Artificial Intelligence" Course: Talks about AI's challenges, societal implications, machine learning, deep learning, generative models and more!

Lesson 1: What is AI?

- Father of AI = Alan Turing
- AI originated as a new branch of knowledge within computer science
- AI vs AGI (Artificial General Intelligence)
    - AI = uses data to complete specific and relatively simpler tasks
        - Perceives, interprets, and learns from data. Reasons and makes decisions
        - Excels at solving specific tasks
        - *Ex of AI: voice assistants, facial recognition, personalized recommendations, autonomous industrial robots*
    - AGI = equal to or exceeding average human intelligence
        - We are only about halfway to AGI in some ways; have not yet reached complete AGI
        - Solves a breadth of tasks intelligently
        - *Ex of AGI: self-driving cars, AlphaGo, generative AI (Large Language Models like GPT)*
- How to map the conception of AI into a real-world scenario and associated workflow with multiple steps logically ordered. Ex:
    - 1. Collect a history of medical data
    - 2. Organize the data and feed it to an Ai system for diabetes diagnosing
    - 3. Do some reasoning over a new patient's data
    - 4. Output a diagnosis for the patient
    - 5. Based on diagnosis results, initiate treatment for the patient

- Things AI can do: predictions and inference (machine learning), pattern recognition (predictions and inference, clustering, anomaly detection, data generation; identify patterns in the data to help make decisions), optimization (find the best possible solution for a problem at a minimum cost, under constraints), automation (follow set of rules to perform repetitive tasks)
    - Automation is not AI, but it can definitely improve AI!
    - Ex of optimization AI problem: AI for smarter traffic lights

- Limitations of AI: emotional intelligence, empathy

- Subdomains of AI
    - Machine Learning: learn from data; predictions, inference
        - Deep learning: neural networks; solve most challenging AI problems
        - Ex: Given credit card transaction data, learn the patterns behind potentially harmful transactions and classify future transactions as secure or not
    - Knowledge representation and reasoning: reason, communicate with other AI systems
        - Ex: Represent concepts and instructions to play chess in a way that the computer can understand and reason about
    - Robotics: act and manipulate physical environment
        - Ex: autonomous assembly of pieces for building a new car
    - Computer vision: visually perceiving objects in the environment
        - Ex: Log in to your personal laptop by placing your finger on its fingerprint detector
    - **Natural Language Processing (NLP): analyze, understand, communicate human language**
        - **Ex: Process an input piece of text in English and automatically translate it into Spanish**

- Related disciplines: Data Science, Math and Statistics, Psychology/Ethics/Law

Lesson 2: Tasks AI can solve

- Algorithms: inputs -> process -> outputs
- AI algorithms: learn by themselves to produce better outputs or processes from input data
- Data acquisition: sensing the environment
    - Transform perceptions into data
    - NLP and audio: capturing speech, sounds
    - CV: satellite images, fingerprint
    - Robotics and sensors: temperature, touch, motion, gravity
- Symbiotic relationship between IoT devices and sensors for environmental data collection and the use of AI to make decisions accordingly
    - Ex: temperature and ultrasonic sensors detect people's presence in rooms of the house and regulate room temperature accordingly
- Structured and unstructured datasets
    - Datasets are no longer just lists of registries following a tabular row-column format. More and more AI solutions can handle data collections in a variety of formats: images, videos, text, audio, etc.
        - Unstructured ex: collection of flower photographies, folder containing hundreds of songs in MP3 format, product reviews written by customers
        - Structured ex: data about products in a supermarket: name, category, stock levels, unit prices
- ML: learn from data and identify patterns to perform inference tasks: predictions, classifications, clustering…
    - Supervised learning: classification, regression (uses labeled input and output data / gets labeled observations with known class a priori)

        - Ex: random forests, decision trees
        - Classification: assign each data observation the category / class it may belong to
        - Regression: assign each data observation a numerical output or a label based on its inputs
        - Time series forecasting: goal = predict the future values of our target, based on observations of the past
- Unsupervised learning: clustering, anomaly detection
    - Unlabeled data
    - Tries to learn properties or patterns behind the data
    - Use clustering techniques to find subgroups of data with similar characteristics (i.e. k-means algorithm)
    - Anomaly detection: detecting abnormal data observations
        - Ex: detecting unusual peaks in the EUR-GBP currency exchange rate
    - Association rule discovery: find common occurrences of items in transaction data
    - Reinforcement learning: learn by experience (trial and error)
- Reinforcement learning: learning from past experiences
- Deep learning: using deep neural network architectures (resembling human brains) to solve more complex variants of these ML problems
    - For particularly advanced + challenging problems
    - All problems solved by ML can also be addressed through deep learning!
        - BUT deep learning can also: recognized objects in images + video, translate + summarize, generative AI (large language models, image and music generation)
- Interacting with the environment
    - CV
        - Image processing: intelligently enhance images and video (think: filters!)
        - Object detection: identify subjects in images/video for surveillance, logistics, etc
        - Motion analysis
        - Image and video generation
    - Robotics
    - **NLP (natural language processing)**
        - **The most popular AI area today (with the permission of deep learning)!**
        - **Text-based**
            - **Text classification**
            - **Sentiment analysis: extract positive and negative feelings in text, e.g. customer reviews**
            - **Question answering (Chatbots)**
            - **Text summarization**
        - Speech-based
            - Text-to-speech
            - Speech-to-text

Lesson 3: Harnessing AI in Organizations

- AI team
    - **Execution and MLOps (machine learning operations)**
        - **AI architects - oversee the AI solution architecture, making decisions like selecting the right tools to use**
        - **Data scientists - analyze complex datasets, prepare the data, train and evaluate ML models, and analyze model outputs**
        - **ML engineers - focus on deploying implemented models into production and managing the AI system infrastructure**
        - **Data engineers - build robust data processing pipelines**
- Cloud-based infrastructure offers flexible resource provisioning and scalability, while on-premises infrastructure demands a fixed hardware investment.
- Measuring success in production
    - AI/ML metrics
        - Model degradation: the measured metric value gets worse over time
    - Business metrics: KPIs
        - Ex: conversion rate, satisfaction (retail); turnaround time (healthcare - time an AI-based medical diagnostic system takes to provide patient diagnoses)
- Risks in AI system: data bias, lack of transparency, ethical concerns, dubious system reliability, vulnerability to cyber threats
    - PoC = Proof-of-Concept: pilot demonstrator to validate feasibility and potential value + early risk identification

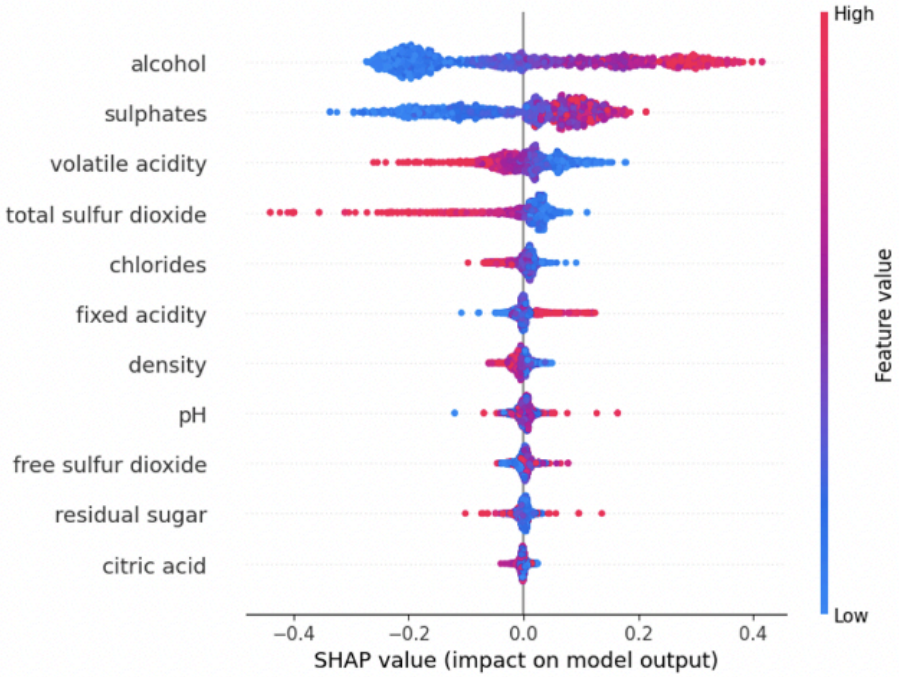
Lesson 4: The human side of AI

- AI democratization
- White-box vs black-box AI systems
    - White-box: transparent and easily interpretable models/systems (ex: decision trees for classification)
    - Black-box: higher complexity, little or no degree of understandability (ex: random forests, deep neural networks)
- **Basic Explainable AI (XAI) tools:**
    - **XAI: methods and tools to increase AI systems and models' transparency and explainability**
        - **Model introspection: examining internal model parameters to understand decisions**
            - Kind of like what we did my first summer with Prof. Sage! Understanding blackbox machine learning models such as random forests
        - Model documentation: shareable architecture and design considerations
        - Model visualization: human-friendly rep of data features and model outputs
        - **Feature importance !!!**
            - **Looking at the impact or contribution of features / predictors in model outputs**
            - **SHAP (SHapley Additive exPlanations)**
                - **Feature importance visualizations toolbox -> might be useful for my senior capstone!**

- **Not only capable of describing the importance of features for a model as a whole, but also the positive and negative relationships between predictor features and the target output**
- **We can even get explanations for a single prediction on a data observation!!**

Below is a **SHAP visualization** of the importance of data features used in a regression model to estimate the quality of red wine samples, using chemical properties (sulfates, alcohol, density, etc.) as predictor attributes or features.

- The **plot** shows the relative importance of predictor attributes (chemical properties) in the inference process applied by the model for estimating the quality of a given observation (red wine sample).
- The **range of feature values** is represented by **colors ranging from blue (lowest) to pink (highest)**. The position of each feature's colored bar concerning the horizontal axis shows the effect feature values have in the resulting model output: some features' values lead to higher outputs when they are higher, while some other features' values lead to lower outputs when they are higher.



Take a deep look at the above SHAP plot, and select which of the following statements about model explainability and feature importance are true :

Answer the question 50XP

Possible Answers

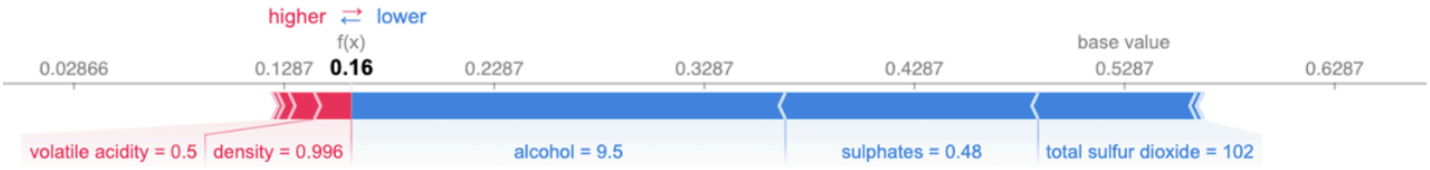
Select all correct answers

- [x] Feature importance is ranked in descending order in the SHAP plot. PRESS 1
- [x] Alcohol is the most influential chemical property of red wine in determining its quality, according to this regression model. PRESS 2
- [ ] Features like chlorides, fixed acidity, density, and pH, have little importance in predicting wine quality and therefore should be discarded and not used by the model. PRESS 3
- [x] Higher values of the alcohol feature have a positive impact in increasing the predicted output for wine quality level. PRESS 4
- [ ] Higher values of the sulphates feature have a negative impact in the predicted output, decreasing the value of wine quality. PRESS 5

Unboxing the SHAP

One of the reasons behind the magic of **XAI (Explainable AI) tools like SHAP**, is the ability to **display not only** the overall **importance of predictor features** behind a model, **but also** the specific importance and **relationship between input features and one specific** model's output or **prediction**.

The following plot depicts the importance played by red wine chemical properties (predictor features) in estimating its quality (output), for a given wine observation (data instance). The resulting model prediction is a **quality level of 0.16** for this wine.



Below are **four statements** related to **feature importance and model behavior in this individual prediction**.

One of these statements is False. Can you find it?

Answer the question 50XP

Possible Answers

Select one answer

- The sulfates value for this wine contributed to pushing the output value (wine quality) lower. PRESS 1
- Alcohol has a significant influence in predicting the quality of this wine, but it contributed to pushing this quality downward. PRESS 2
- Total sulfur dioxide is the most influential feature in the prediction. PRESS 3
- Despite its low value and small feature importance, the volatile acidity contributed to pushing the resulting prediction higher. PRESS 4

- Bias in AI systems examples
  - Screening job resumes
    - Biased training data: mostly male
    - Unfair treatment of female candidates
    - Solutions: active data collection, bias-correction algorithms
  - E-commerce recommendations
    - Popular products are overly promoted
    - New or different products are disregarded
    - Solutions: techniques and metrics for diverse and fair recommendations

Week 4: Completed “Intro to Python” DataCamp Course and Learned About Gradient Descent!

“Introduction to Python” Course: An Intro to the Basics of Python + a Prereq for “Working with the OpenAI API” and “Intro to NLP in Python”

Section 1: Python Basics

- Variables and data types
    - int
    - float
    - string
    - bool

Section 2: Python Lists

- Lists in python: fam = [a, b, c, d]
- Subsetting lists
    - The first element in the list has index 0
    - [start: end] (end is exclusive)
    - Negative indexing
        - listname[-1] - last element of the list
- Examples of slicing and dicing
    - Use slicing to create a list, `downstairs`, that contains the first 6 elements of `areas`:

```
downstairs = areas[:6]
```

    - Do a similar thing to create a new variable, `upstairs`, that contains the last 4 elements of `areas`.

```
upstairs = areas[6:]
```

        - Key takeaway: If you don't specify the `begin` index, Python figures out that you want to start your slice at the beginning of your list. If you don't specify the `end` index, the slice will go all the way to the last element of your list.
- In order to make an explicit copy of a (and not reference the same) list, use list() or [:]
    - Ex: areas_copy = list(areas)
    - Ex 2: areas_copy = areas[:] - this copies all elements in areas

Section 3: Functions

- Functions = piece of reusable code that solves a particular task
- Call function instead of having to write code yourself
- Built-in functions: max(), round(), help(functionName), print(), type(), str(), int(), bool(), float()
- Methods: Functions that belong to objects
    - Everything = object
    - Objects have methods associated, depending on type
- Packages = directory of Python Scripts
    - Each script = module
    - numpy - arrays
    - Matplotlib - data viz
    - scikit-learn - machine learning
    - For installing Python packages:
        - In Terminal: python3 get-pip.py
        - pip3 install numpy

        - Import numpy as np

Section 4: NumPy (Numeric Python)

- Different types: different behavior!
    - Adding arrays -> element-wise sum
- 2D NumPy Arrays

```
# Import numpy package
import numpy as np

# Create np_baseball (2 cols)
np_baseball = np.array(baseball)

# Print out the 50th row of np_baseball
print(np_baseball[49,:])

# Select the entire second column of np_baseball: np_weight_lb
np_weight_lb = np_baseball[:,1]

# Print out height of 124th player
print(np_baseball[123,0])
```

- Arguments for np.random.normal()
    - Distribution mean
    - Distribution standard deviation
    - Number of samples

```
# Print out correlation between first and second column. Replace 'None'
corr = np.corrcoef(np_baseball[:,0],np_baseball[:,1])
print("Correlation: " + str(corr))
```

<u>Gradient Descent</u>

- Gradient Descent, Step-by-Step: https://www.youtube.com/watch?v=sDv4f4s2SB8
    - A method for unconstrained mathematical optimization
    - A first-order iterative algorithm for finding a local minimum of a differentiable multivariate function
    - Idea: Take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent
        - Stepping in the direction of the gradient will conversely lead to a local maximum of that function (gradient ascent)
    - Useful in ML and deep learning models for minimizing the cost or loss function
        - Goal: find the minimum of a cost function by iteratively adjusting the model parameters
    - Summary of steps:
        - 1) Take the derivative of Loss Function (i.e. sum of squared residuals) for each parameter in it. In fancy ML terms, take the gradient of the loss function.
            - The cost or loss function measures how well a model is performing by quantifying the difference between predicted and actual values

            - Goal: minimize cost function to improve accuracy of model
        - 2) Pick random values for the parameters.
        - 3) Plug the parameter values into the derivatives (ahem, the gradient).
            - The gradient is a vector that points in the direction of the steepest increase of the function at a specific point. In gradient descent, it refers to the partial derivatives of the cost function with respect to each model parameter
            - The gradient indicates the direction in which the parameters should be adjusted to decrease the cost
        - **4) Calculate the step sizes: Step Size = Slope * Learning Rate**
            - The gradient descent can be very sensitive to the learning rate. Luckily, in practice, a reasonable learning rate can be determined automatically by starting large and getting smaller with each step.
        - 5) Calculate the new parameters: **New Parameter = Old Parameter - Step Size**
        - Return to Step 3 and repeat until Step Size is very small, or you reach the Maximum Number of Steps (predefined)
            - Gradient descent involves iteratively updating the model parameters in the opposite direction of the gradient.
            - **The update rule is given by: $\theta=\theta-\alpha\nabla J(\theta)$ where θ represents the model parameters, α is the learning rate (a hyperparameter that determines the size of the steps taken during optimization), and $\nabla J(\theta)$ is the gradient of the cost function with respect to the parameters**
    - Stochastic Gradient Descent - uses a randomly selected subset of the data at every step rather than the full dataset in order to reduce the time spent calculating the derivatives of the loss function
        - Con: makes it more noisy

Week 9: Got started with a course called “Introduction to Natural Language Processing in Python” which talks about how to identify and separate words, how to extract topics in a text, and how to build your own fake news classifier

<u>“Introduction to Natural Language Processing in Python” Course:</u>

Lesson 1: Regular expressions and word tokenization

- NLP = field of study focused on making sense of language using stats and computers
    - Basics of NLP
        - Topic identification
        - Text classification
    - NLP application ex: chatbots, translation, sentiment analysis, etc
- Regular expressions (regex)
    - Strings with a special syntax
    - Allow us to match patterns in other strings
    - Applications of regular expressions:
        - Find all web links in a doc
        - Parse email addresses
        - Remove/replace unwanted characters

- **Pattern first, string second!!!**
    - Common regex patterns
        - \w+ = word
        - \d = digit
        - \s = space
        - .* = wildcard
        - + or * = greedy match
        - \S = not space
        - [a-z] = lowercase group
    - Ex: re.split('\s+', 'Split on spaces.') returns ['Split', 'on', 'spaces.']

```
# Write a pattern to match sentence endings: sentence_endings
sentence_endings = r"[.?!]"

# Split my_string on sentence endings and print the result
print(re.split(sentence_endings, my_string))

# Find all capitalized words in my_string and print the result
capitalized_words = r"[A-Z]\w+"
print(re.findall(capitalized_words, my_string))

# Split my_string on spaces and print the result
spaces = r"\s+"
print(re.split(spaces, my_string))

# Find all digits in my_string and print the result
digits = r"\d+"
print(re.findall(digits, my_string))
```

- Tokenization
    - Turning a string or document into tokens (smaller chunks)
    - One step in preparing a text for NLP
    - Many different theories and rules; you can create your own rules using regular expressions
        - Ex: breaking out words or sentences separating punctuation, separating all hashtags in a tweet
    - nltk: natural language toolkit
    - Why tokenize?
        - Easier to map part of speech
        - Matching common words
        - Removing unwanted tokens
    - Types of tokenizers:
        - word_tonenize: break a string into word tokens
        - sent_tokenize: tokenize a document into sentences
        - regexp_tokenize: tokenize a string or document based on a regular expression pattern
        - TweetTokenizer: special class just for tweet tokenization, allowing you to separate hashtags, mentions and lots of exclamation points!!!
    - Difference between re.search() and re.match()

- **If you want to find a pattern that might not be at the beginning of a string, use SEARCH.**
- **Match tries to match a string from the beginning until it cannot match any longer. If you want to be specific about the initial pattern, use MATCH.**

```
# Import necessary modules
from nltk.tokenize import sent_tokenize, word_tokenize

# Split scene_one into sentences: sentences
sentences = sent_tokenize(scene_one)

# Use word_tokenize to tokenize the fourth sentence: tokenized_sent
tokenized_sent = word_tokenize(sentences[3])

# Make a set of unique tokens in the entire scene: unique_tokens
unique_tokens = set(word_tokenize(scene_one))

# Print the unique tokens result
print(unique_tokens)
```

- Regex groups using or “|”
  - OR is represented using |
  - You can define a group using ()
  - You can define explicit character ranges using []
- Regex ranges and groups
  - [A-Za-z]+ = upper and lowercase English alphabet
  - [0-9] = numbers from 0 to 9
  - [A-Za-z\-\.]+ = upper and lowercase English alphabet, - and .
    - Backwards slashes (escape characters) tell us to look for a hyphen or a period
  - (a-z) = a, - and z
  - (\s + |,) = spaces or a comma

```
# Tokenize and print all words in german_text
all_words = word_tokenize(german_text)
print(all_words)

# Tokenize and print only capital words
capital_words = r"[A-Z|Ü]\w+"
print(regexp_tokenize(german_text, capital_words))

# Tokenize and print only emoji
emoji =
"['\U0001F300-\U0001F5FF'|'\U0001F600-\U0001F64F'|'\U0001F680-\U0001F6FF'|'\u2600-\u26FF\u2700-\u27BF']"
print(regexp_tokenize(german_text, emoji))
# Split the script into lines: lines
lines = holy_grail.split('\n')
```

```
# Replace all script lines for speaker
pattern = "[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
lines = [re.sub(pattern, '', l) for l in lines]

# Tokenize each line: tokenized_lines
tokenized_lines = [regexp_tokenize(s, "\w+") for s in lines]

# Make a frequency list of lengths: line_num_words
line_num_words = [len(t_line) for t_line in tokenized_lines]

# Plot a histogram of the line lengths
plt.hist(line_num_words)

# Show the plot
plt.show()
```

Lesson 2: Simple Topic Identification (Using basic NLP models, you will identify topics from texts based on term frequencies. You'll experiment and compare two simple methods: **bag-of-words and Tf-idf using NLTK,** and a new library Gensim)

- Word counts with bag-of-words
    - Basic method for finding topics in a text
    - Need to first create tokens using tokenization + then count up all the tokens
    - The more frequent a word, the more important it might be
    - Can be a great way to determine the significant words in a text!
    - Ex bag of words mapping:
        - ('The', 2), ('box', 2), ('.', 2), ('cat', 2), ('is', 1), ('in', 1), ('the', 1)
- Building a bag-of-words counter:

```
# Import Counter
from collections import Counter

# Tokenize the article: tokens
tokens = word_tokenize(article)

# Convert the tokens into lowercase: lower_tokens
lower_tokens = [t.lower() for t in tokens]

# Create a Counter with the lowercase tokens: bow_simple
bow_simple = Counter(lower_tokens)

# Print the 10 most common tokens
print(bow_simple.most_common(10))
```

- Simple text preprocessing
    - Helps make for better input data
    - Lemmatization/stemming
        - Shorten words to their room stems
    - Removing stop words (which don't carry a lot of meaning, such as as or the), punctuation, or unwanted tokens
    - Good to experiment with different approaches

- tokens = [w for w in word_tokenize(text.lower())
  if w.isalpha()]
  - The is.alpha() method will return alphabetic strings only (removing punctuation + numbers)
- No_stops = [t for t in tokens
  if t not in stopwords.words('english')]
- Counter(no_stops).most_common(2)

- Text preprocessing practice:

```
from nltk.stem import WordNetLemmatizer

# Retain alphabetic words: alpha_only
alpha_only = [t for t in lower_tokens if t.isalpha()]

# Remove all stop words: no_stops
no_stops = [t for t in alpha_only if t not in english_stops]

# Instantiate the WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

# Lemmatize all tokens into a new list: lemmatized
lemmatized = [wordnet_lemmatizer.lemmatize(t) for t in no_stops]

# Create the bag-of-words: bow
bow = Counter(lemmatized)

# Print the 10 most common tokens
print(bow.most_common(10))
```

- Introduction to gensim
  - Gensim = popular open-source NLP library
  - Uses top academic models to perform complex tasks
    - Building document or word vectors
    - Performing topic identification and document comparison
  - What is a word vector? - chart in the video isn't displaying
    - Word vectors = multi-dimensional mathematical representations of words created using deep learning methods. They give us insight into relationships between words in a corpus.
  - Corpus = a set of texts used to help perform natural language processing tasks
  - Gensim models can be easily saved, updated, reused
    - Our dictionary can also be updated
    - A more advanced and feature rich bag-of-words

- Creating my first gensim dictionary and corpus:

```
# Import Dictionary
from gensim.corpora.dictionary import Dictionary

# Create a Dictionary from the articles: dictionary
dictionary = Dictionary(articles)
```

```
# Select the id for "computer": computer_id
computer_id = dictionary.token2id.get("computer")

# Use computer_id with the dictionary to print the word
print(dictionary.get(computer_id))

# Create a MmCorpus: corpus
corpus = [dictionary.doc2bow(article) for article in articles]

# Print the first 10 word ids with their frequency counts from the fifth document
print(corpus[4][:10])
```

- Using this new gensim corpus and dictionary to see the most common terms per document and across all documents:
  - Note that …
    - `defaultdict` allows us to initialize a dictionary that will assign a default value to non-existent keys. By supplying the argument `int`, we are able to ensure that any non-existent keys are automatically assigned a default value of `0`. This makes it ideal for storing the counts of words in this exercise.
    - `itertools.chain.from_iterable()` allows us to iterate through a set of sequences as if they were one continuous sequence. Using this function, we can easily iterate through our `corpus` object (which is a list of lists).

```
# Save the fifth document: doc
doc = corpus[4]

# Sort the doc for frequency: bow_doc
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)

# Print the top 5 words of the document alongside the count
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)

# Create the defaultdict: total_word_count
total_word_count = defaultdict(int)
for word_id, word_count in itertools.chain.from_iterable(corpus):
    total_word_count[word_id] += word_count

# Create a sorted list from the defaultdict: sorted_word_count
sorted_word_count = sorted(total_word_count.items(), key=lambda w: w[1],
reverse=True)

# Print the top 5 words across all documents alongside the count
for word_id, word_count in sorted_word_count[:5]:
    print(dictionary.get(word_id), word_count)
```

- Tf-idf with gensim
  - **Tf-idf = Term frequency - inverse document frequency**
  - Allows you to determine the most important words in each document
  - **Each corpus may have shared words beyond just stopwords**

- **These words should be down-weighted in importance**
- Example from astronomy: “Sky” - want to downweight this word
- Ensures most common words don’t show up as key words!
- **In brief, Tf-idf keeps document specific frequent words weighted high and the common words across the entire corpus weighted low**

Tf-idf formula

$$w_{i,j} = tf_{i,j} * \log(\frac{N}{df_i})$$

$$w_{i,j} = \text{tf-idf weight for token } i \text{ in document } j$$

$$tf_{i,j} = \text{number of occurences of token } i \text{ in document } j$$

$$df_i = \text{number of documents that contain token } i$$

$$N = \text{total number of documents}$$

- Unpacking the formula a bit:
  - The weight will be low if the term does not appear often in the document because the tf variable will then be low.
  - The weight will also be low if the log is close to 0. Recall: log(1) = 0. If total number of documents / number of documents that have the term is close to 1, our log will be close to 0. Thus, words that occur across many or all documents will have a very low tf-idf weight.
  - If the word occurs only in a few documents, that log will return a higher number!
- Example of computing tf-idf:
  - Calculate the tf-idf weight for the word “computer,” which appears 5 times in a doc containing 100 words. Given a corpus containing 200 documents, with 20 documents mentioning the word “computer”, tf-idf can be calculated by multiplying the term frequency with inverse document frequency.
    - Term frequency = % share of the word compared to all tokens in the document
    - Inverse document frequency = log of the total number of documents in a corpora divided by the number of documents containing the term
    - Thus, the tf-idf weight in this case can be calculated as follows: (5/100) * log(200/20)
- Determining new significant terms for my corpus by applying gensim’s tf-idf:

```
# Create a new TfidfModel using the corpus: tfidf
tfidf = TfidfModel(corpus)

# Calculate the tfidf weights of doc: tfidf_weights
```

```
tfidf_weights = tfidf[doc]

# Print the first five weights
print(tfidf_weights[:5])

# Sort the weights from highest to lowest: sorted_tfidf_weights
sorted_tfidf_weights = sorted(tfidf_weights, key=lambda w: w[1],
reverse=True)

# Print the top 5 weighted words
for term_id, weight in sorted_tfidf_weights[:5]:
    print(dictionary.get(term_id), weight)
```

Lesson 3: Named-Entity Recognition (Learn how to identify the who, what, and where of your texts using pre-trained models on English and non-English text. Also learn how to use some new libraries, polyglot, and spaCy, to add to your NLP toolbox)

- **Named Entity Recognition = NLP task to identify important named entities in the text**
    - **People, places, organizations**
    - **Dates, states, works of art, etc!**
- Can be used alongside topic identification or on its own
- Who? What? When? Where?
- The Stanford CoreNLP library:
    - Integrated into Python via nltk
    - Java based
    - Support for NER as well as coreference and dependency trees

Using nltk to find the named entities in the article:

```
# Tag each tokenized sentence into parts of speech:
pos_sentences
pos_sentences = [nltk.pos_tag(sent) for sent in token_sentences]

# Create the named entity chunks: chunked_sentences
chunked_sentences = nltk.ne_chunk_sents(pos_sentences, binary =
True)

# Test for stems of the tree with 'NE' tags (NE = named entity)
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, "label") and chunk.label() == "NE":
            print(chunk)
```

Using extracted named entities and their groupings from a series of newspaper articles to chart the diversity of named entity types in the articles:

```
# Create the defaultdict: ner_categories
```

```
ner_categories = defaultdict(int)

# Create the nested for loop
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label'):
            ner_categories[chunk.label()] += 1

# Create a list from the dictionary keys for the chart labels:
labels
labels = list(ner_categories.keys())

# Create a list of the values: values
values = [ner_categories.get(v) for v in labels]

# Create the pie chart
plt.pie(values, labels=labels, autopct='%1.1f%%',
startangle=140)

# Display the chart
plt.show()
```

- Q: When using the Stanford library with NLTK, what is needed to get started?
    - A: NLTK, the Stanford Java Libraries and some environment variables to help with integration
- Intro to SpaCy
    - **SpaCy = NLP library similar to gensim, with different implementations**
    - **Focus on creating NLP pipelines to generate models and corpora**
    - **Open-source, with extra libraries and tools**
        - Displacy - a visualization tool for viewing parse trees which uses Node-js to create interactive text
    - **Why use SpaCy for NER?**
        - Easy pipeline creation
        - **Different entity types compared to nltk - often labels entities differently**
        - **Informal language corpora**
            - **Allows you to easily find entities in Tweets and chat messages**
        - **Quickly growing library! Might even have more languages supported very soon**

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

nlp.entity

doc = nlp(“””Berlin is the capital of Germany; and the residence of Chancellor Angela Merkel.”””)
doc.ents

_____

(GPE = geopolitical entity)

- Using the same text we used in the first exercise of this chapter, look at the results using spaCy's NER annotator. How do they compare?

```
# Import spacy
import spacy

# Instantiate the English model: nlp
nlp = spacy.load('en_core_web_sm', disable = ['tagger','parser','matcher'])

# Create a new document: doc
doc = nlp(article)

# Print all of the found entities and their labels
for ent in doc.ents:
   print(ent.label_, ent.text)

Output:
ORG Apple
PERSON Travis Kalanick of Uber
PERSON Tim Cook
ORG Apple
CARDINAL Millions
LOC Silicon Valley
ORG Yahoo
PERSON Marissa Mayer
MONEY 186
```

- **Q: Which are the *extra* categories that `spacy` uses compared to `nltk` in its named-entity recognition?**
    - **NORP, CARDINAL, MONEY, WORK OF ART, LANGUAGE, EVENT**
    - Recall: nltk also has GPE (geopolitical entity) and PERSON types
- Multilingual NER (Named Entity Recognition) with polyglot
    - NLP library which uses word vectors
    - **Why polyglot?**
        - **Vectors for many different languages - more than 130!**
        - **Can use it for transliteration (= the ability to translate text by swapping characters from one language to another)**

_____

from polyglot.text import Text
text = “”” _____ “””
ptext = Text(text)

ptext.entities

- Entity types
  - I-ORG = organization
  - I-LOC = location
  - I-PER = person
- May experience some duplication of entity labels

```
# Create a new text object using Polyglot's Text class: txt
txt = Text(article)

# Print each of the entities found
for ent in txt.entities:
   print(ent)

# Print the type of ent
print(type(ent))

# Create the list of tuples: entities
entities = [(ent.tag, ' '.join(ent)) for ent in txt.entities]

# Print entities
print(entities)
```

- Spanish NER with polyglot: this is a more blog-like text. The Text object has been created as txt, and each entity has been printed. Determine how many of the entities contain the words “Márquez” or “Gabo” - these refer to the same person in different ways!

```
# Initialize the count variable: count
count = 0

# Iterate over all the entities
for ent in txt.entities:
   # Check whether the entity contains 'Márquez' or 'Gabo'
   if("Márquez" in ent or "Gabo" in ent):
       # Increment count
       count += 1

# Print count
print(count)

# Calculate the percentage of entities that refer to "Gabo": percentage
percentage = count / len(txt.entities)
print(percentage)
```

Lesson 4: Classifying Fake News using Supervised Learning with NLP

- What is supervised learning?
  - Form of machine learning
    - Problem has predefined training data
    - This data has a label (or outcome) you want the model to learn

        - Classic example: Fischer's Iris Data
            - Classification problem
            - Goal: Make good hypotheses about the species based on geometric features
- Supervised learning with NLP
    - **Need to use language instead of geometric features**
        - **To help create features and train a model, we will use scikit-learn, a powerful open-source library**
    - **How to create supervised learning data from text?**
        - **Use bag-of-words models of tf-idf (term frequency-in document frequency) as features**
        - Example: IMDB movie dataset - full of movie plots and genres
            - Goal: predict movie genre (action or sci-fi) based on plot summary
            - Categorical features generated using preprocessing
- Supervised learning steps
    - Collect and preprocess our data
    - Determine a label (example: movie genre)
    - Split data into training and test sets, keeping them separate so we can build our model using only the training data.
        - The test data remains unseen so we can test how well our model performs after it is trained.
    - Extract features from the text to help predict the label
        - Bag-of-words vector built into scikit-learn
    - Evaluate trained model using the test set
        - Other methods for evaluating model performance: k-fold cross validation (see ML course on DataCamp)
- **Q: Which of the following are possible features for a text classification problem?**
    - **A: All of the above (number of words in doc, specific named entities, AND language)**
- Building word count vectors with scikit-learn
    - Predicting movie genre
        - Dataset consisting of movie plots + corresponding genre
        - **Goal: Create bag-of-word vectors for the movie plots**
            - **Can we predict genre based on the words used in the plot summary?**

```
import pandas as pd

from sklearn.model_selection import train_test_split
```

from sklearn.feature_extraction.text import CountVectorizer
df = … # Load data into DataFrame
y = df[‘Sci-Fi’]
X_train, X_test, y_train, y_test = train_test_split(df[‘plot’], y, test_size = 0.33, random_state = 53)
# random_state is similar to random seed and ensures that we have a repeatable result)
count_vectorizer = CountVectorizer(stop_words = ‘english’) # turns my text into bag of words vectors similar to a Gensim corpus, it will also remove English stop words from the movie plot summaries as a preprocessing step
count_train = count_vectorizer.fit_transform(X_train.values) # fit_transform is a handy shortcut which will call the model’s fit and then transform methods, generating a mapping of words with IDs and vectors representing how many times each word appears in the plot
count_test = count_vectorizer.transform(X_test.values)
print(count_vectorizer.get_feature_names()[:10]) # print the first 10 features of the count_vectorizer

---

- Similar to the CountVectorizer created in the previous exercise, you will work on creating tf-idf vectors for your documents. You will set up a TfidfVectorizer and investigate some of its features:

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

# Initialize a TfidfVectorizer object: tfidf_vectorizer
tfidf_vectorizer = TfidfVectorizer(stop_words = "english", max_df = 0.7)

# Transform the training data: tfidf_train
tfidf_train = tfidf_vectorizer.fit_transform(X_train.values)

# Transform the test data: tfidf_test
tfidf_test = tfidf_vectorizer.transform(X_test.values)

# Print the first 10 features
print(tfidf_vectorizer.get_feature_names()[:10])

# Print the first 5 vectors of the tfidf training data
print(tfidf_train.A[:5])
```

- To get a better idea of how the vectors work, investigate them by converting them into pandas DataFrames:

```
# Create the CountVectorizer DataFrame: count_df
count_df = pd.DataFrame(count_train.A, columns=count_vectorizer.get_feature_names())

# Create the TfidfVectorizer DataFrame: tfidf_df
tfidf_df = pd.DataFrame(tfidf_train.A, columns=tfidf_vectorizer.get_feature_names())

# Print the head of count_df
print(count_df.head())

# Print the head of tfidf_df
```

```
print(tfidf_df.head())

# Calculate the difference in columns: difference
difference = set(count_df.columns) - set(tfidf_df.columns)
print(difference)

# Check whether the DataFrames are equal
print(count_df.equals(tfidf_df))
```

- Training and testing a classification model with scikit-learn:
    - N**aive Bayes classifier**
        - **Naive Bayes Model**
            - **Commonly used for testing NLP classification problems**
            - Basis in probability
        - **Given a particular piece of data, how likely is a particular outcome?**
        - Examples:
            - If the plot has a spaceship, how likely is it to be sci-fi?
            - Given a spaceship and an alien, how likely now is it sci-fi?
        - **Each word from CountVectorizer acts as a feature**
        - Naive Bayes: simple and effective
            - HOWEVER: Does not work well with floats, such as tfidf weighted inputs. Instead, use support vector machines, or even linear models!

---

from sklearn.naive_bayes import MultinomialNB **(recall: from packages import module)**
from sklearn import metrics # the metrics module is for evaluating model performance
nb_classifier = MultinomialNB() # initialize our class

nb_classifier.fit(count_train, y_train) # determines the internal parameters based on the dataset, pass the training count vectorizer first and the training labels second
pred = nb_classifier.predict(count_test)
metrics.accuracy_score(y_test, pred) # calculate percentage of correct genre guesses / total guesses

metrics.confusion_matrix(y_test, pred, labels = [0,1]) # takes in test labels, predictions, and a list of labels

---

- Diagonal of a confusion matrix = true scores
- **Q: Which is the most reasonable model to use when training a new supervised model using text vector data?**
    - **A: Naive Bayes!**
- Train and test a “fake news” Naive Bayes model using the CountVectorizer data. The training and test sets have been created, and count_vectorizer, count_train, and count_test have been computed:

```
# Import the necessary modules
from sklearn import metrics
from sklearn.naive_bayes import MultinomialNB

# Instantiate a Multinomial Naive Bayes classifier: nb_classifier
```

```
nb_classifier = MultinomialNB()

# Fit the classifier to the training data
nb_classifier.fit(count_train, y_train)

# Create the predicted tags: pred
pred = nb_classifier.predict(count_test)

# Calculate the accuracy score: score
score = metrics.accuracy_score(y_test, pred)
print(score)

# Calculate the confusion matrix: cm
cm = metrics.confusion_matrix(y_test, pred, labels = ['FAKE','REAL'])
print(cm)
```

- Simple NLP, Complex Problems
    - Problems with translation
        - Ex: the German text has many different words related to economics that are all simply closest to the english vector economics
    - Problems with sentiment analysis
        - Complex issues include snark / sarcasm and difficult problems with negation (for example: “I liked it BUT it could have been better”)
        - Active research regarding how separate communities use the same words differently
    - Problems with language biases
        - Language can contain its own prejudices and unfair treatment towards groups
        - When we train word vectors on these prejudiced texts, our word vectors will likely reflect those problems (ex: “He’s a professor. She’s a babysitter.”)
- Q: What are possible next steps you could take to improve the “fake news” model you have created:
    - A: All of the above (tweak alpha levels, try a new classification model, train on a larger dataset, and improve text preprocessing)!
- Improving your model exercise: Test a few different alpha levels using the Tfidf vectors to determine if there is a better performing combination. The training and test sets have been created, and tfidf_vectorizer, tfidf_train, and tfidf_test have been computed:

```
# Create the list of alphas: alphas
alphas = np.arange(0,1,0.1)

# Define train_and_predict()
def train_and_predict(alpha):
    # Instantiate the classifier: nb_classifier
    nb_classifier = MultinomialNB(alpha = alpha)
    # Fit to the training data
    nb_classifier.fit(tfidf_train, y_train)
    # Predict the labels: pred
    pred = nb_classifier.predict(tfidf_test)
```

```
    # Compute accuracy: score
    score = metrics.accuracy_score(y_test, pred)
    return score

# Iterate over the alphas and print the corresponding score
for alpha in alphas:
    print('Alpha: ', alpha)
    print('Score: ', train_and_predict(alpha))
    print()

<script.py> output:
    Alpha:  0.0
    Score:  0.8813964610234337

    Alpha:  0.1
    Score:  0.8976566236250598

    Alpha:  0.2
    Score:  0.8938307030129125

    Alpha:  0.30000000000000004
    Score:  0.8900047824007652

    Alpha:  0.4
    Score:  0.8857006217120995

    Alpha:  0.5
    Score:  0.8842659014825442

    Alpha:  0.6000000000000001
    Score:  0.874701099952176

    Alpha:  0.7000000000000001
    Score:  0.8703969392635102

    Alpha:  0.8
    Score:  0.8660927785748446

    Alpha:  0.9
    Score:  0.8589191774270684
```

- ^^^ The best alpha is 0.1!
- Now that you have built a “fake news” classifier, investigate what it has learned. You can map the important vector weights back to actual words using some simple inspection techniques. You have your well performing tfidf Naive BAyes classifier available as nb_classifier, and the vectors as tfidf_vectorizer:

```
# Get the class labels: class_labels
class_labels = nb_classifier.classes_
```

```
# Extract the features: feature_names
feature_names = tfidf_vectorizer.get_feature_names()

# Zip the feature names together with the coefficient array and sort by weights:
feat_with_weights
feat_with_weights = sorted(zip(nb_classifier.coef_[0], feature_names))

# Print the first class label and the top 20 feat_with_weights entries
print(class_labels[0], feat_with_weights[:20])

# Print the second class label and the bottom 20 feat_with_weights entries
print(class_labels[1], feat_with_weights[-20:])
```

```
FAKE [(-12.641778440826338, '0000'), (-12.641778440826338, '000035'),
(-12.641778440826338, '0001'), (-12.641778440826338, '0001pt'), (-12.641778440826338,
'000km'), (-12.641778440826338, '0011'), (-12.641778440826338, '006s'),
(-12.641778440826338, '007'), (-12.641778440826338, '007s'), (-12.641778440826338,
'008s'), (-12.641778440826338, '0099'), (-12.641778440826338, '00am'),
(-12.641778440826338, '00p'), (-12.641778440826338, '00pm'), (-12.641778440826338,
'014'), (-12.641778440826338, '015'), (-12.641778440826338, '018'),
(-12.641778440826338, '01am'), (-12.641778440826338, '020'), (-12.641778440826338,
'023')]
REAL [(-6.790929954967984, 'states'), (-6.765360557845787, 'rubio'),
(-6.751044290367751, 'voters'), (-6.701050756752027, 'house'), (-6.695547793099875,
'republicans'), (-6.670191249042969, 'bush'), (-6.661945235816139, 'percent'),
(-6.589623788689861, 'people'), (-6.559670340096453, 'new'), (-6.489892292073902,
'party'), (-6.452319082422527, 'cruz'), (-6.452076515575875, 'state'),
(-6.397696648238072, 'republican'), (-6.376343060363355, 'campaign'),
(-6.324397735392007, 'president'), (-6.2546017970213645, 'sanders'),
(-6.144621899738043, 'obama'), (-5.756817248152807, 'clinton'), (-5.596085785733112,
'said'), (-5.357523914504495, 'trump')]
```