# Source code:

**1.** Basic Tic-Tac-Toe implementation:

#import random module for generating random posiitons for the other player

import random

#Define the indicies of the tic-tac-toe board positions

board = [0,1,2,3,4,5,6,7,8]

#draw the board

def showboard():

print (str(board[0])+" | "+str(board[1])+" | "+str(board[2]))

print ('----------')

print (str(board[3])+" | "+str(board[4])+" | "+str(board[5]))

print ('----------')

print (str(board[6])+" | "+str(board[7])+" | "+str(board[8]))

#Define a function where it returns 'True' if any of the game possibilities

#consists of the same marker (i.e., same player). This function helps in

#defining who's the winner

def CheckLine(char, spot1, spot2, spot3):

if board[spot1] == char and board[spot2] == char and board[spot3] == char:

return True

#Defining the winning possibilities

def CheckAll(char):

#winning rows

if CheckLine(char, 0, 1, 2):

return True

if CheckLine(char, 3, 4, 5):

return True

if CheckLine(char, 6, 7, 8):

return True

#winning columns

if CheckLine(char, 0, 3, 6):

return True

if CheckLine(char, 1, 4, 7):

return True

if CheckLine(char, 2, 5, 9):

return True

#winning diagonals

if CheckLine(char, 2, 4, 6):

return True

if CheckLine(char, 0, 4, 8):

return True

#The game Code

#NOTE

#While loops, like the ForLoop, are used for repeating sections of code -

#but unlike a for loop, the while loop will not run n times, but until a

#defined condition is met. As the for loop in Python is so powerful, while is

#rarely used, except in cases where a user's input is required.

while True: # Having True as a condition ensures that the following code runs

#until it's broken by one of the three conditions: X wins, O Wins or spot is

#taken.

#input a number to fill in one of the board position

number = input("Select a number between 0-8: ")

number = int(number)

#check, if the position that is chosen in the previous step is not occupied by

#either x or o, then put x in it

if board[number] != 'x' and board[number] != 'o':

board[number] = 'x'

#keep checking when all the posiiotn have the same marker (x) until the first

# player is the winner!

if CheckAll('x') == True:

print ("X Wins!")

break;#This break belongs to having the first player

#is the winner

#we need another while loop to generate random input for the second

#player. This loop is broken when 'if CheckAll('o') == True' is met.

while True:

number2 = random.randint(0,8) #generate random numbers from 0 to 8

#check the availability of the position

if board[number2] != 'x' and board[number2] != 'o':

board[number2] = 'o'

showboard()

#if the three posititions have o, then the second player is the

#winner

if CheckAll('o') == True:

print ("O Wins!")

break; #This break belongs to having the second player

#is the winner

break; #belongs to checking the availibility for the second player

#' if board[number2] != 'x' and board[number2] != 'o':'

else: #if the input is taken by another player (this can be checked by

#board[number]!='x' and board[number]!='o'), then do the following line

print ("Spot Taken... Try again.")

break; #belongs to the first 'if'

**2.** Case Study 1: Caesar Cypher implementation:

#TODO: Create a dictionary letters with keys consisting of the numbers from 0

#to 26, and values consisting of the lowercase letters of the English alphabet,

# including the space ' ' at the end.

# Let's look at the lowercase letters.

import string

string.ascii\_lowercase

# We will consider the alphabet to be these letters, along with a space.

alphabet = string.ascii\_lowercase + " "

# create `letters` here!

letters = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e', 5:'f', 6:'g', 7:'h', 8:'i'

, 9:'j', 10:'k', 11:'l', 12:'m', 13:'n', 14:'o', 15:'p', 16:'q'

, 17:'r', 18:'s', 19:'t', 20:'u', 21:'v', 22:'w', 23:'x', 24:'y'

, 25:'z', 26:' '}

#Another way is:

letters = {i:alphabet[i] for i in range(0,27)}

print(letters)

#alphabet and letters are already defined. Create a dictionary encoding with

#keys being the characters in alphabet and values being numbers from 0-26,

#shifted by an integer encryption\_key=3. For example, the key a should have

#value encryption\_key, key b should have value encryption\_key + 1, and so on.

#If any result of this addition is less than 0 or greater than 26, you can

#ensure the result remains within 0-26 using result % 27.

encryption\_key = 3

# define `encoding` here!

encoding = {alphabet[i]:((i + encryption\_key) % 27) for i in range(27)}

#OR

#alphabet\_len = len(alphabet)

#encoding = {alphabet[i]:((i + encryption\_key) % alphabet\_len) for i in range(alphabet\_len)}

print(encoding)

message = "hi my name is caesar"

def caesar(message, key):

# return the encoded message as a single string!

alphabet\_len = len(alphabet)

encoded\_message = {alphabet[i]:((i + key) % alphabet\_len) for i in range(alphabet\_len)}

return ''.join([alphabet[encoded\_message[letter]] for letter in message])

encoded\_message = caesar(message, 3)

print(encoded\_message)

#Decode and save coded\_message using caesar and encryption\_key = -3.

#coded\_message is already loaded from the previous problem.

#Store your decoded message as decoded\_message.

#Print decoded\_message. Does this recover your original message?

decoded\_message = caesar(encoded\_message, -3)

print(decoded\_message)

#It can be seen that it is the same original message!

**3.** Case Study 2: Word Frequency distribution in Hamlet:

from collections import Counter

text='Romeo and Juliet'

def count\_words\_fast(text):

"""count the number of times each word occurs in text (str).

Return dictionary where keys are unique words and values are

word counts. skip punctuations"""

text = text.lower() #lowercase for the counting letters so the function can cont the same words whether it's capatilised or not

skips = [".", ",", ";", ":", "'", '"'] #skipping all the punctuations to not be counted with the words that come bfore them

for ch in skips:

text = text.replace(ch,"")

word\_counts = Counter(text.split(" "))

return word\_counts

def read\_book(title\_path):

"""Read a book and return it as a string"""

with open(title\_path, "r") as current\_file:

text = current\_file.read()

text = text.replace("\n","").replace("\r","")

return text

def word\_stats(word\_counts):

"""return the number of unique words and word frequencies"""

num\_unique = len(word\_counts) #calculate the number of unique words in the text

counts = word\_counts.values() #calculate the frequency of each word in the text

return(num\_unique,counts)

#TODO: Write a function word\_count\_distribution(text) that takes a book string and returns a dictionary with items

#corresponding to the count of times a collection of words appears in the translation, and values corresponding to

#the number of number of words that appear with that frequency.

#TODO: First use count\_words\_fast(text) to create a dictionary called word\_counts with unique words in the dictionary

#as keys and their frequency in the book as values.

#TODO: Next, create and return a new dictionary count\_distribution with unique values from word\_counts as keys and their

#frequency as values. For example, 'you are what you eat' contains three words that occur once and one word that occurs twice,

#so word\_count\_distribution('you are what you eat') should return a dictionary {1:3, 2:1}.

def word\_count\_distribution(text):

word\_counts = count\_words\_fast(text)

count\_distribution = dict(Counter(word\_counts.values()))

return count\_distribution

#TODO: 'Romeo and Juliet' is preloaded as text. Call word\_count\_distribution(text), and save the result as distribution.

distribution = word\_count\_distribution(text)

print(distribution)

#TODO: Create a function more\_frequent(distribution) that takes a word frequency dictionary (like that made in Exercise 1)

# and outputs a dictionary with the same keys as those in distribution

# (the number of times a group of words appears in the text), and values corresponding to the fraction of words that occur

# with more frequency than that key.

import numpy as np

def more\_frequent(distribution):

counts = list(distribution.keys())

frequency\_of\_counts = list(distribution.values())

cumulative\_frequencies = np.cumsum(frequency\_of\_counts)

more\_frequent = 1 - cumulative\_frequencies / cumulative\_frequencies[-1] #To obtain the fraction of words more frequent than this,

#divide this cumulative sum by its maximum, and subtract this value from 1.

return dict(zip(counts, more\_frequent))

#TODO: Call more\_frequent(distribution).

more\_frequent(distribution)

#Edit the code used to read though each of the books in our library, and store

#the word frequency distribution for each translation of William Shakespeare's

#"Hamlet" as a Pandas dataframe hamlets with columns named "language" and

#"distribution". word\_count\_distribution is preloaded from Exercise 1.

#How many translations are there? Which languages are they translated into?

import pandas as pd

hamlets = pd.DataFrame(columns=('language','distribution'))

book\_dir = "Books"

title\_num = 1

for language in book\_titles:

for author in book\_titles[language]:

for title in book\_titles[language][author]:

if title == "Hamlet":

inputfile = data\_filepath+"Books/"+language+"/"+author+"/"+title+".txt"

text = read\_book(inputfile)

distribution = word\_count\_distribution(text)

hamlets.loc[title\_num] = language, distribution

title\_num += 1

#There are three translations: English, German, and Portuguese.

#Plot the word frequency distributions of each translations on a single log-log plot.

#Note that we have already done most of the work for you. Do the distributions of each translation differ?

import matplotlib.pyplot as plt

colors = ["crimson", "forestgreen", "blueviolet"]

handles, hamlet\_languages = [], []

for index in range(hamlets.shape[0]):

language, distribution = hamlets.language[index+1], hamlets.distribution[index+1]

dist = more\_frequent(distribution)

plot, = plt.loglog(sorted(list(dist.keys())),sorted(list(dist.values()),

reverse = True), color = colors[index], linewidth = 2)

handles.append(plot)

hamlet\_languages.append(language)

plt.title("Word Frequencies in Hamlet Translations")

xlim = [0, 2e3]

xlabel = "Frequency of Word $W$"

ylabel = "Fraction of Words\nWith Greater Frequency than $W$"

plt.xlim(xlim); plt.xlabel(xlabel); plt.ylabel(ylabel)

plt.legend(handles, hamlet\_languages, loc = "upper right", numpoints = 1)

plt.show()

**4.** Case Study 3: Wine Classification:

#TODO: Read in the data as a pandas dataframe using pd.read\_csv.

#The data can be found at https://s3.amazonaws.com/demo-datasets/wine.csv.

import pandas as pd

import numpy as np

data = pd.read\_csv('https://s3.amazonaws.com/demo-datasets/wine.csv') #read directly from the link

#TODO: The dataset remains stored as data. Two columns in data are is\_red and color,

#which are redundant. Drop color from the dataset, and save the new dataset as

#numeric\_data.

numeric\_data = data.drop('color', axis=1)

print (numeric\_data.head(5))

#TODO: To ensure that each variable contributes equally to the kNN classifier,

#we need to standardize the data. First, from each variable in numeric\_data,

#subtract its mean. Second, for each variable in numeric\_data, divide by its

#standard deviation. Store your standardized result as numeric\_data.

numeric\_data = (numeric\_data - np.mean(numeric\_data)) / np.std(numeric\_data)

print(numeric\_data.head(5))

#TODO: Principal component analysis is a way to take a linear snapshot of the data

#from several different angles, with each snapshot ordered by how well it

#aligns with variation in the data. The sklearn.decomposition module contains

#the PCA class, which determines the most informative principal components of

#the data (a matrix with columns corresponding to the principal components).

#Use pca.fit(numeric\_data).transform(numeric\_data) to extract the first two

#principal components from the data. Store this as principal\_components.

import sklearn.decomposition

pca = sklearn.decomposition.PCA(n\_components=2)

principal\_components = pca.fit(numeric\_data).transform(numeric\_data)

#TODO: The first two principal components can be accessed using principal\_components[:,0]

#and principal\_components[:,1]. Store these as x and y respectively, and plot the

#first two principal components. The high and low quality wines will be colored

#using red and blue. How well are the two groups of wines separated by the first

#two principal components?

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

from matplotlib.backends.backend\_pdf import PdfPages

observation\_colormap = ListedColormap(['red', 'blue'])

x = principal\_components[:,0]

y = principal\_components[:,1]

plt.plot(principal\_components[:,0], principal\_components[:,1])

plt.title("Principal Components of Wine")

plt.scatter(x, y, alpha = 0.2,

c = data['high\_quality'], cmap = observation\_colormap, edgecolors = 'none')

plt.xlim(-8, 8); plt.ylim(-8, 8)

plt.xlabel("Principal Component 1"); plt.ylabel("Principal Component 2")

plt.show()

#TODO: We are now ready to fit the wine data to our kNN classifier.

#Create a function accuracy(predictions, outcomes) that takes two lists of the

#same size as arguments and returns a single number, which is the percentage of

#elements that are equal for the two lists.

def accuracy(predictions, outcomes):

for i in predictions:

for j in outcomes:

if i==j:

Percentage = 100\*np.mean(predictions == outcomes)

return Percentage

#TODO: Use accuracy to compare the percentage of similar elements in

#x = np.array([1,2,3]) and y = np.array([1,2,4]).

x = np.array([1,2,3])

y = np.array([1,2,4])

percentage = accuracy(x, y)

#TODO: Print your answer.

print (percentage)

#TODO: The dataset remains stored as data. Because most wines in the dataset are

#classified as low quality, one very simple classification rule is to predict

#that all wines are of low quality. Use the accuracy function (preloaded into

#memory as defined in Exercise 5) to calculate how many wines in the dataset

#are of low quality. Accomplish this by calling accuracy with 0 as the first

#argument, and data["high\_quality"] as the second argument.

number\_of\_low\_quality = accuracy(0, data["high\_quality"])

#TODO: Print your result.

print (number\_of\_low\_quality)

#TODO: Use knn.predict(numeric\_data) to predict which wines are high and low quality

#and store the result as library\_predictions.

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n\_neighbors = 5)

knn.fit(numeric\_data, data['high\_quality'])

library\_predictions = knn.predict(numeric\_data)

#TODO: Use accuracy to find the accuracy of your predictions, using library\_predictions

#as the first argument and data["high\_quality"] as the second argument.

predictions\_accuracy = accuracy(library\_predictions, data["high\_quality"])

#TODO: Print your answer. Is this prediction better than the simple classifier

#in Exercise 6?

print (predictions\_accuracy) # Yes, this is better!

#TODO: Unlike the scikit-learn function, our homemade kNN classifier does not take

#any shortcuts in calculating which neighbors are closest to each observation,

#so it is likely too slow to carry out on the whole dataset. To circumvent this,

#fix the random generator using random.seed(123), and select 10 rows from the

#dataset using random.sample(range(n\_nrows), 10). Store this selection as selection.

import random

n\_rows = data.shape[0]

random.seed(123)

selection = random.sample(range(n\_rows), 10)

#TODO: The sample of 10 row indices are stored as selection from the previous exercise.

#For each predictor p in predictors[selection],

#use knn\_predict(p, predictors[training\_indices,:], outcomes, k=5) to predict

#the quality of each wine in the prediction set, and store these predictions

#as a np.array called my\_predictions. Note that knn\_predict is already defined

#as in the Case 3 videos.

predictors = np.array(numeric\_data)

training\_indices = [i for i in range(len(predictors)) if i not in selection]

outcomes = np.array(data["high\_quality"])

my\_predictions = [knn\_predict(p, predictors[training\_indices,:], outcomes, k=5) for p in predictors[selection]]

#TODO: Using the accuracy function, compare these results to the selected rows from

#the high\_quality variable in data using my\_predictions as the first argument

#and data.high\_quality[selection] as the second argument. Store these results

#as percentage.

percentage = accuracy(my\_predictions, data.high\_quality[selection] )

#TODO: Print your answer.

print(percentage)

**5.** Case Study 4: Scotch Whisky Analysis Using Bokeh

from bokeh.models import HoverTool, ColumnDataSource

import numpy as np

# Let's plot a simple 5x5 grid of squares, alternating in color as red and blue.

plot\_values = [1,2,3,4,5]plot\_colors = ["red", "blue"]

# How do we tell Bokeh to plot each point in a grid? Let's use a function that

# finds each combination of values from 1-5.

from itertools import product

grid = list(product(plot\_values, plot\_values))

print(grid)

#Let's create the names and colors we will use to plot the correlation matrix

#of whisky flavors. Later, we will also use these colors to plot each distillery

#geographically. Create a dictionary region\_colors with regions as keys

#and cluster\_colors as values.

cluster\_colors = ["red", "orange", "green", "blue", "purple", "gray"]

regions = ["Speyside", "Highlands", "Lowlands", "Islands", "Campbelltown", "Islay"]

region\_colors = dict(zip(regions,cluster\_colors))

#Print region\_colors.

print(region\_colors)

#correlations is a two-dimensional np.array with both rows and columns

#corresponding to distilleries and elements corresponding to the flavor

#correlation of each row/column pair. Let's define a list correlation\_colors,

#with string values corresponding to colors to be used to plot each distillery

#pair. Low correlations among distillery pairs will be white, high correlations

#will be a distinct group color if the distilleries from the same group, and

#gray otherwise. Edit the code to define correlation\_colors for each distillery

#pair to have input 'white' if their correlation is less than 0.7.

#whisky.Group is a pandas dataframe column consisting of distillery group

#memberships. For distillery pairs with correlation greater than 0.7, if they

#share the same whisky group, use the corresponding color from cluster\_colors.

#Otherwise, the correlation\_colors value for that distillery pair will be

#defined as 'lightgray'.

distilleries = list(whisky.Distillery)

correlation\_colors = []

for i in range(len(distilleries)):

for j in range(len(distilleries)):

if correlations[i,j] < .70: # if low correlation,

correlation\_colors.append('white') # just use white.

else: # otherwise,

if whisky.Group[i] == whisky.Group[j]: # if the groups match,

correlation\_colors.append(cluster\_colors[whisky.Group[i]]) # color them by their mutual group.

else: # otherwise

correlation\_colors.append('lightgray') # color them lightgray.

#We will edit the following code to make an interactive grid of the correlations

#among distillery pairs using correlation\_colors and correlations.

#correlation\_colors is a list of each distillery pair. To convert correlations

#from a np.array to a list, we will use the flatten method. Define the color

#of each rectangle in the grid using to correlation\_colors.

#Define the alpha (transparency) values using correlations.flatten().

#Define correlations and using correlations.flatten(). When the cursor hovers

#over a rectangle, this will output the distillery pair, show both distilleries

#as well as their correlation coefficient.

source = ColumnDataSource(

data = {

"x": np.repeat(distilleries,len(distilleries)),

"y": list(distilleries)\*len(distilleries),

"colors": correlation\_colors,

"alphas": correlations.flatten(),

"correlations": correlations.flatten(),

}

)

output\_file("Whisky Correlations.html", title="Whisky Correlations")

fig = figure(title="Whisky Correlations",

x\_axis\_location="above", tools="resize,hover,save",

x\_range=list(reversed(distilleries)), y\_range=distilleries)

fig.grid.grid\_line\_color = None

fig.axis.axis\_line\_color = None

fig.axis.major\_tick\_line\_color = None

fig.axis.major\_label\_text\_font\_size = "5pt"

fig.xaxis.major\_label\_orientation = np.pi / 3

fig.rect('x', 'y', .9, .9, source=source,

color='colors', alpha='alphas')

hover = fig.select(dict(type=HoverTool))

hover.tooltips = {

"Whiskies": "@x, @y",

"Correlation": "@correlations",

}

show(fig)

#Next, we provide an example of plotting points geographically.

#Run the following code, to be adapted in the next section.

#Compare this code to that used in plotting the distillery correlations.

points = [(0,0), (1,2), (3,1)]

xs, ys = zip(\*points)

colors = ["red", "blue", "green"]

output\_file("Spatial\_Example.html", title="Regional Example")

location\_source = ColumnDataSource(

data={

"x": xs,

"y": ys,

"colors": colors,

}

)

fig = figure(title = "Title",

x\_axis\_location = "above", tools="resize, hover, save")

fig.plot\_width = 300

fig.plot\_height = 380

fig.circle("x", "y", 10, 10, size=10, source=location\_source,

color='colors', line\_color = None)

hover = fig.select(dict(type = HoverTool))

hover.tooltips = {

"Location": "(@x, @y)"

}

show(fig)

#Adapt the given code from the beginning to show(fig) in order to define a

#function location\_plot(title, colors). This function takes a string title

# and a list of colors corresponding to each distillery and outputs a Bokeh

#plot of each distillery by latitude and longitude. As the cursor hovers over

#each point, it displays the distillery name, latitude, and longitude.

def location\_plot(title, colors):

output\_file(title+".html")

location\_source = ColumnDataSource(

data={

"x": whisky[" Latitude"],

"y": whisky[" Longitude"],

"colors": colors,

"regions": whisky.Region,

"distilleries": whisky.Distillery

}

)

fig = figure(title = title,

x\_axis\_location = "above", tools="resize, hover, save")

fig.plot\_width = 400

fig.plot\_height = 500

fig.circle("x", "y", 10, 10, size=9, source=location\_source,

color='colors', line\_color = None)

fig.xaxis.major\_label\_orientation = np.pi / 3

hover = fig.select(dict(type = HoverTool))

hover.tooltips = {

"Distillery": "@distilleries",

"Location": "(@x, @y)"

}

show(fig)

#whisky.Region is a pandas column containing the regional group membership for

#each distillery. Make a list consisting of the value of region\_colors for

#each distillery, and store this list as region\_cols.

region\_cols = [region\_colors[i] for i in list(whisky["Region"])]

#Use location\_plot to plot each distillery, colored by its regional grouping.

location\_plot("Whisky Locations and Regions", region\_cols)

#Use list comprehensions to create the list region\_cols consisting of the color

# in region\_colors that corresponds to each whisky in whisky.Region.

region\_cols = [region\_colors[i] for i in whisky['Region']]

#Similarly, create a list classification\_cols consisting of the color in

#cluster\_colors that corresponds to each cluster membership in whisky.Group.

classification\_cols = [cluster\_colors[j] for j in whisky['Group']]

#location\_plot remains stored from the previous exercise. Use it to create two

#interactive plots of distilleries, one colored by defined region called

#region\_cols and the other with colors defined by coclustering designation

#called classification\_cols. How well do the coclustering groupings match the

#regional groupings?

location\_plot("Whisky Locations and Regions", region\_cols)

location\_plot("Whisky Locations and Groups", classification\_cols)