Nicholas Clement

103151912

# Problem 1:

Key = BOULDER

Method used to break the Vigenère cipher was the Kasiski examination. The Kasiski examination allows us to deduce the length of the keyword. The key to determining the length of the keyword lies in repeating substrings. Here are some python functions I wrote to find all of the repetitions in the cipher text, and then compare the different spacing of these substrings. Finally I check which substrings repeat every n spaces, and based on the highest value of n determine the key length.

---------------------------------------------------------------------------------------------------------------

def findReps(cipher):

'''Find repeating sequences of characters in our cipher text,

return them in a Counter'''

occurances = Counter()

x = 0

while len(cipher)>0:

for letter in range(len(cipher)):

x+=1

occurances[cipher[0:letter+1]]+=1

print("Iteration #", x)

cipher = cipher[1:] #remove a letter from cipher

mostCommon = occurances.most\_common()

nGrams = []

for each in mostCommon:

# print(each)

if len(each[0]) >=3 and len(each[0]) <=10 and each[1] == 3:

nGrams.append(each)

# print(nGrams)

return (nGrams)

def findLocations(cipher,chars):

'''given a cipher and the repeating substrings, find the indexes

that the repetitions occur at'''

locations = []

index = start = 0

while start>=0:

if start == 0:

start = cipher.find(chars, start)

else:

start = cipher.find(chars, start)

if start == -1:

return locations

locations.append(start)

start += len(chars)

# print(locations)

return(locations)

def repLocations(cipher):

'''Gather the occurance location for each set of repeating chars,

then store that in a dictionary'''

reps = findReps(cipher)

occuranceInfo= {}

for each in reps:

occuranceInfo[each[0]] = findLocations(cipher, each[0])

return occuranceInfo

def modOccurances(occuranceInfo):

'''Take the contents returned from repLocations and use it

to find the different spacing of collisions for all repeating patterns.

Do this using the mod function to determine the keylength

(factor) that is most common between all repitions.'''

keys = range(3,21)

dictSpace = {}

for key in keys:

listSpaces = []

for each in occuranceInfo:

for index in occuranceInfo[each]:

if int(index)%int(key) == 0 and (index>0):

listSpaces.append(index)

dictSpace[key] = len(listSpaces)

return dictSpace

---------------------------------------------------------------------------------------------------------------

Now since we know the length of the key (in our case 7) we know that we can use every nth index of our cipher text as a Caesar cipher. This is where frequency analysis comes into play, by knowing the frequencies of letters in each of the 7 Cesar ciphers we can compare to the relative frequency of the English alphabet. My attempted implementation of frequency analysis worked partially, I was able to deduce that the key looked like boulder although my program didn’t come to that conclusion on its own. Here is the code I wrote for this part:

---------------------------------------------------------------------------------------------------------------

def frequencyAnalysis(texty, keyLength):

knownFreq = {'A': 8.167, 'B': 1.492, 'C': 2.782, 'D': 4.253, 'E': 12.702, 'F': 2.228, 'G': 2.015,

'H': 6.094, 'I': 6.996, 'J': 0.153, 'K': 0.772, 'L': 4.025, 'M': 2.406, 'N': 6.749,

'O': 7.507, 'P': 1.929, 'Q': 0.095, 'R': 5.987, 'S':6.327, 'T': 9.056, 'U': 2.758,

'V': 0.978, 'W': 2.360, 'X': 0.150, 'Y': 1.974, 'Z': 0.074}

occuranceInfo = {}

for key in range(keyLength):

occurances = []

for (intLetter, letter) in enumerate(texty):

if intLetter % keyLength == key:

occurances.append(letter)

occuranceInfo[key] = occurances

# print(occuranceInfo)

for common in sorted(knownFreq, key=knownFreq.get, reverse=False):

for key in range(keyLength):

tempTexty = list(texty)

print("".join(tempTexty))

currentOccur = Counter(occuranceInfo[key])

currentOccur = currentOccur.most\_common(1)

# print(currentOccur)

print(currentOccur[0][0])

for (intLetter, letter) in enumerate(texty):

if intLetter % keyLength == key:

shiftBy = (ord(common) - ord(currentOccur[0][0]))

# g = 7 e = 5 26 - (5-1)

# shiftBy = ord(common) - ord(currentOccur[0][0])

order = (ord(letter) + shiftBy)

print("Letters are:", currentOccur[0][0], common, "shifting by", shiftBy)

if order>ord('Z'): #in the case of order being greater than z

order = -1 + ord('A') +((order) - ord('Z'))

elif order < ord('A'):

order = 1+ord('Z') - (ord('A') - order)

print("Replaced", tempTexty[intLetter], "with", chr(order))

tempTexty[intLetter] = chr(order) #replace old char with new char

# print(str(tempTexty))

print(detect\_langs(str("".join(tempTexty))))

if 'en' in detect\_langs(str("".join(tempTexty))) and detect\_langs(str("".join(tempTexty)))['en'] > 0.5:

print("Detection Rate =", detect\_langs(str("".join(tempTexty))), "\n\n")

print("".join(tempTexty))

return

else:

print("Detection Rate =", detect\_langs(str("".join(tempTexty))), "\n\n")

---------------------------------------------------------------------------------------------------------------

You may notice that I used the function detect\_langs, this comes from the python langdetect package that is ported from googles language detection abilities. Another approach I used didn’t focus on frequency analysis, it attempted to brute force the solution by taking the Cartesian products of the different possible shifts of all 7 sub-ciphers. Here is that code:

---------------------------------------------------------------------------------------------------------------

def bruteForce(texty, keyLength):

x = range(0,26)

z=(product(x,repeat=keyLength))

x = 0

tempTexty = texty

for layout in z:

print("Iteration Number:",x)

x+=1

for (intLetter, letter) in enumerate(texty):

for (intshiftBy, shiftBy) in enumerate(layout):

if (intLetter % keyLength) == intshiftBy:

order = (ord(letter) + shiftBy)

if order>ord('Z'): #in the case of order being greater than z

order = ord('A') +((ord(letter)+shiftBy) - ord('Z'))

tempTexty[intLetter].replace(tempTexty[intLetter], chr(order)) #replace old char with new char

detectionRate=detect(tempTexty)

if detectionRate == "en":

print("Success!!!")

print(detectionRate)

print(tempTexty)

return(tempTexty)

return(100)

---------------------------------------------------------------------------------------------------------------

Sadly this program would take days to run to completion, with there being 26 possible shifts for all 7 ciphers we end up with 26^7 possible answers to check, leaving us with a little more than eight billion possible plaintext solutions, all of which need to be checked to determine if they are English.

# Problem 2:

part\_a\_var\_english=10.405667735207103

part\_b\_var\_plaintext=0.0000000