Team: Bakhtiyar Rakhimzhanov, Nazira Tukeyeva

Group: BD-1903

Github Link of Bakhtiyar: <https://github.com/Godadoreu>

Github Link of Nazira: <https://github.com/nazirait>

Github repository for this task: <https://github.com/Godadoreu/Assignment-7-IT>

**Assignment 7 Report**

|  |  |
| --- | --- |
| Task Part | Responsible team member |
| Part 1 Function (Reading txt file, sequence of bit checker, technique for decoding (7,4)) | Nazira |
| Part 2 Function ( Error Generator, technique for decoding (16,11)) | Bakhtiyar |
| Assignment Report, Hamming decoding | Both |

General description:

There are 6 functions, where two of them were used from previous assignments.

Step 1. In order to read and separate the strings the decode\_text and separator function are used.

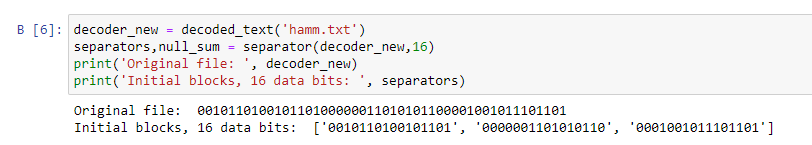
**Code:**

decoder\_new = decoded\_text('hamm.txt')

separators,null\_sum = separator(decoder\_new,16)

print('Original file: ', decoder\_new)

print('Initial blocks, 16 data bits: ', separators)



Step 2. Selecting the 30-50% of blocks from bits and generate error at 30-50 percent probability.

**Code:**

def ErrorGen(bitstring, percent = None):

'''Generating the error at 30-50% probability rate'''

import numpy as np

result = []

## If the percent is not specified

if percent == None:

# it will give 30,40,50 percent probability by default

percents = round(np.random.choice([3,4,5],replace = False)/10 \* len(bitstring))

result = []

# if the number of percent is rich the limit, loop will break

for x in range(percents):

count = 0

if count==percents:

break

else:

# choosing random bitstring from block

randoms = np.random.choice(bitstring,replace = False)

# getting the index of random bitstring

index = bitstring.index(randoms)

# doing the list

list\_randoms = list(randoms)

length = len(list\_randoms)

# choosing the random place, where the error will be placed

rand\_place = np.random.choice(length,1)[0]

# if the value is equal to 0, it will be 1 and vice versa

if list\_randoms[rand\_place] == '0':

list\_randoms[rand\_place] = '1'

elif list\_randoms[rand\_place] == '1':

list\_randoms[rand\_place] = '0'

# joining into one string

lists = ''.join(list\_randoms)

# append the result to list

result.append(lists)

# replacing the old block with error one

bitstring[index] = lists

count+=1

else:

# if the percent specified, getting the number of blocks with errors

percentage = percent/100

percents = round(percentage \* len(bitstring))

result = []

for x in range(percents):

count = 0

if count==percents:

break

else:

randoms = np.random.choice(bitstring,replace = False)

list\_randoms = list(randoms)

index = bitstring.index(randoms)

length = len(list\_randoms)

rand\_place = np.random.choice(length,1)[0]

if list\_randoms[rand\_place] == '0':

list\_randoms[rand\_place] = '1'

elif list\_randoms[rand\_place] == '1':

list\_randoms[rand\_place] = '0'

lists = ''.join(list\_randoms)

result.append(lists)

bitstring[index] = lists

count+=1

# returning the error list and the new list with errors

return result,bitstring

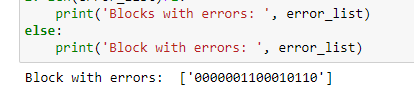
error\_list,new\_list = ErrorGen(separators,30)

if len(error\_list)>1:

print('Blocks with errors: ', error\_list)

else:

print('Block with errors: ', error\_list)



Step 3. Function HammingDecode, where we are going to make decoding of hamming code in 16,11 extended version and saving it into new file.

**Code:**

def HammingDecode(bits):

'''Function for decoding the bits from hamming code'''

resultant = []

# making loop of blocks

for x in bits:

bites = list(x)

print(bites)

checker = []

count\_err = 0

not\_checker = []

print('Checking parity bits: ')

# getting the tuple of each parity bit

checker\_8 = (9,10,11,12,13,14,15,8)

checker\_1 = (1,3,5,7,9,11,13,15)

checker\_2 = (3,6,7,10,11,14,15,2)

checker\_4 = (5,6,7,12,13,14,15,4)

# making calculations using the XOR

value\_1 = int(bites[3])^int(bites[5])^int(bites[7])^int(bites[9])^int(bites[11])^int(bites[13])^int(bites[15])

#print(value\_1)

value\_2 = int(bites[3])^int(bites[6])^int(bites[7])^int(bites[10])^int(bites[11])^int(bites[14])^int(bites[15])

#print(value\_2)

value\_4 = int(bites[5])^int(bites[6])^int(bites[7])^int(bites[12])^int(bites[13])^int(bites[14])^int(bites[15])

#print(value\_4)

value\_8 = int(bites[9])^int(bites[10])^int(bites[11])^int(bites[12])^int(bites[13])^int(bites[14])^int(bites[15])

#print(value\_8)

# getting the count of 1s

value\_0 = bites[1:16].count('1')%2

# checking whether the values are matching

if value\_1 == int(bites[1]):

print('p1: b3 + b5 + b7 + b9 + 11 + 13 + 15 = ', bites[3], '+', bites[5], '+', bites[7], '+', bites[9], '+', bites[11],

'+', bites[13], '+', bites[15], ' = ', bites[1], ' correct')

# if it is correct appending the index positions to list

checker.extend(checker\_1)

else:

print('p1: b3 + b5 + b7 + b9 + 11+13 + 15 = ', bites[3], '+', bites[5], '+', bites[7], '+', bites[9], '+', bites[11],

'+', bites[13], '+', bites[15], ' = ', bites[1], ' incorrect')

# if not correct append the index positions to list

not\_checker.extend(checker\_1)

# count the error per block

count\_err+=1

if value\_2 == int(bites[2]):

print('p2: b3 + b6 + b7 +b10 + b11+b14+b15 = ', bites[3], '+', bites[6], '+', bites[7], '+', bites[10], '+', bites[11],

'+', bites[14], '+', bites[15],' = ', bites[2], ' correct')

checker.extend(checker\_2)

else:

print('p2: b3 + b6 + b7 +b10 + b11+b14+b15 = ', bites[3], '+', bites[6], '+', bites[7], '+', bites[10], '+', bites[11],

'+', bites[14], '+', bites[15],' = ', bites[2], ' incorrect')

not\_checker.extend(checker\_2)

count\_err+=1

if value\_4 == int(bites[4]):

print('p4: b5 + b6 + b7 + b12 + b13 + b14 + b15= ', bites[5], '+', bites[6], '+', bites[7], '+', bites[12],

'+', bites[13], '+', bites[14], '+', bites[15], ' = ', bites[4], ' correct')

checker.extend(checker\_4)

else:

print('p4: b5 + b6 + b7 + b12 + b13 + b14 + b15= ', bites[5], '+', bites[6], '+', bites[7], '+', bites[12],

'+', bites[13], '+', bites[14], '+', bites[15], ' = ', bites[4], ' incorrect')

not\_checker.extend(checker\_4)

count\_err+=1

if value\_8 == int(bites[8]):

print('p8: b9 + b10 + b11 + b12 + b13 + b14 + b15 = ', bites[9], '+', bites[10], '+', bites[11], '+', bites[12], '+',

bites[13],'+', bites[14],'+', bites[15], ' = ', bites[8], ' correct')

checker.extend(checker\_8)

else:

print('p8: b9 + b10 + b11 + b12 + b13 + b14 + b15 = ', bites[9], '+', bites[10], '+', bites[11], '+', bites[12], '+',

bites[13],'+', bites[14],'+', bites[15], ' = ', bites[8], ' incorrect')

not\_checker.extend(checker\_8)

count\_err+=1

# getting only unique list values

lists = sorted(list(dict.fromkeys(checker)))

# if the length of list is 15(without 0 index)

if len(lists)==15:

# check the value of 0 index

if value\_0 == int(bites[0]):

# if they match

print('No error')

#get the initial decoded string

result = bites[3]+bites[5]+bites[6]+bites[7]+bites[9]+bites[10]+bites[11]+bites[12] + bites[13]+bites[14] + bites[15]

print('Decoded bitstring: ', result)

# append the result to the final list

resultant.append(result)

print('\n')

# if the value of 0th index doesn't match

elif value\_0 != int(bites[0]):

# error is on 0th position

print('Error at position: 0')

# if the initial value is 0, turn it to 1 and vice versa

if bites[0]=='1':

bites[0] = '0'

else:

bites[0] = '1'

# show corrected version

print('Corrected bitstring: ',bites)

# do the decoding

result=bites[3]+bites[5]+bites[6]+bites[7]+bites[9]+bites[10]+bites[11]+bites[12] + bites[13]+bites[14] + bites[15]

print('Decoded bitstring: ', result)

# append the result to final list

resultant.append(result)

print('\n')

else:

# if the length is not 15

error\_list = []

#print(not\_checker)

# get the loop of error list

for x in not\_checker:

# if the error list value is not in list append it to another list

if x not in lists:

error\_list.append(x)

print(error\_list)

# get the loop of error list

for x in error\_list:

# count the frequency of each index

val = error\_list.count(x)

# if the frequency match the error num per block

if val == count\_err:

# get the index, as index and break loop

index = x

break

print('Error at position: ', index)

# if the value is 1, replace it with 0 and vice versa

if bites[index]=='1':

bites[index] = '0'

else:

bites[index] = '1'

value\_0 = bites[1:16].count('1')%2

#print(bites)

if value\_0 == int(bites[0]):

print('Corrected bitstring: ',bites)

result=bites[3]+bites[5]+bites[6]+bites[7]+bites[9]+bites[10]+bites[11]+bites[12] + bites[13]+bites[14] + bites[15]

print('Decoded bitstring: ', result)

resultant.append(result)

print('\n')

# returning the final list

return resultant

decode\_version = HammingDecode(new\_list)

# saving the final version in new file

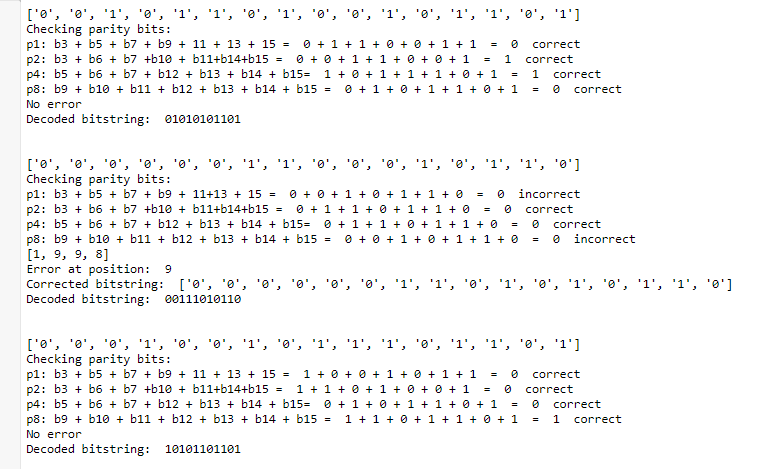
string = ''.join(decode\_version)

file = open('decode\_hamm.txt','w')

file.write(string)

file.close()

decode\_version



Step 4. Function HammingDecode, where we are going to make decoding of hamming code in 8,4 extended version and saving it into new file.

**Code:**

def HammingDecode2(bits):

result = []

for x in bits:

error = []

right = []

count\_err = 0

bites = list(x)

print(bites)

val1 = int(bites[3]) ^ int(bites[5]) ^ int(bites[7])

truth1 = (val1 == int(bites[1]))

val2 = int(bites[3]) ^ int(bites[6]) ^ int(bites[7])

truth2 = (val2 == int(bites[2]))

val4 = int(bites[5]) ^ int(bites[6]) ^ int(bites[7])

truth4 = (val4 == int(bites[4]))

#blocks[0] = sum(blocks[1:])

if int(truth1)>0:

print('p1: b3 + b5 + b7 = ', bites[3], '+', bites[5], '+', bites[7], ' = ', bites[1],' ,correct')

right.extend((1,3,5,7))

else:

print('p1: b3 + b5 + b7 = ', bites[3], '+', bites[5], '+', bites[7], ' = ', bites[1],' ,incorrect')

error.extend((1,3,5,7))

count\_err+=1

if int(truth2)>0:

print('p2: b3 + b6 + b7 = ', bites[3], '+', bites[6], '+', bites[7], ' = ', bites[2], ' ,correct')

right.extend((2,3,6,7))

else:

print('p2: b3 + b6 + b7 = ', bites[3], '+', bites[6], '+', bites[7], ' = ', bites[2], ' ,incorrect')

error.extend((2,3,6,7))

count\_err+=1

if int(truth4)>0:

print('p3: b5 + b6 + b7 = ', bites[6], '+', bites[5], '+', bites[7], ' = ', bites[4], ' ,correct')

right.extend((4,6,5,7))

else:

print('p3: b5 + b6 + b7 = ', bites[6], '+', bites[5], '+', bites[7], ' = ', bites[4], ' ,incorrect')

error.extend((4,6,5,7))

count\_err+=1

uniques = list(dict.fromkeys(right))

#print(uniques)

if len(uniques) == 7:

val0 = bites[1:].count('1')%2

truth0 = (val0==int(bites[0]))

if int(truth0)>0:

print('p0: b1 + b2 + b3 + b4 + b5 + b6 + b7 = ', bites[1], '+', bites[2], '+', bites[3], '+', bites[4], '+',

bites[5],'+', bites[6],'+', bites[7], ' = ', bites[0], ' ,correct')

print('No error')

bitstring = bites[3] + bites[5] + bites[6] + bites[7]

print('Decoded bitstring: ', bitstring)

result.append(bitstring)

print('\n')

else:

print('p0: b1 + b2 + b3 + b4 + b5 + b6 + b7 = ', bites[1], '+', bites[2], '+', bites[3], '+', bites[4], '+',

bites[5],'+', bites[6],'+', bites[7], ' = ', bites[0], ' ,incorrect')

if bites[0]=='1':

bites[0] = '0'

else:

bites[0] = '1'

print('Error in position: 0')

print('Corrected bitstring: ', bites)

bitstring = bites[3] + bites[5] + bites[6] + bites[7]

print('Decoded bitstring: ', bitstring)

result.append(bitstring)

print('\n')

else:

error\_pos = []

index = 0

#print(error)

for x in error:

if x not in uniques:

error\_pos.append(x)

for x in error\_pos:

val = error\_pos.count(x)

if val == count\_err:

index = x

#print(error\_pos)

print('Error at position: ', index)

if bites[index] == '1':

bites[index] = '0'

else:

bites[index] = '1'

val0 = bites[1:].count('1')%2

#print(val0)

if val0 == int(bites[0]):

print('Corrected bitstring: ', bites)

bitstring = bites[3] + bites[5] + bites[6] + bites[7]

print('Decoded bitstring: ', bitstring)

result.append(bitstring)

print('\n')

else:

print('Error in code')

return result

decodes = HammingDecode2(news)

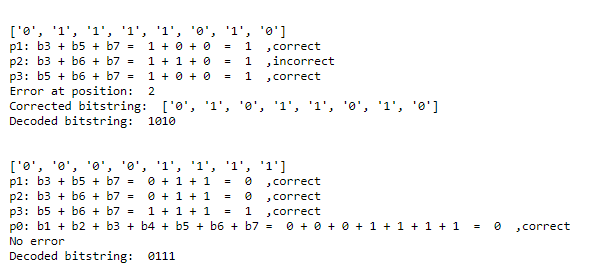
string = ''.join(decode\_version)

files = open('decode\_hamm74.txt','w')

files.write(string)

files.close()

decodes



Step 4. Function checker which helps to check whether our decoded version is equal to the initial version from assignment 3.

**Code:**

def checker(decode\_string,result3):

'''The function which check for the matching of strings'''

print('Decoded sequence: ')

result\_string = ''.join(decode\_string)

# getting the difference of decoded string with initial string

diff = len(result\_string) - len(result3)

# if it is bigger than 0

if diff>0:

# get the string at specific position

result\_string = result\_string[:(len(result\_string)-diff)]

print(result\_string)

print('Sequence from Assignment 3: ')

print(result3)

# checking for matching

if result\_string==result3:

print('They match.')

