Task 1: Substitution ciphers

In this task, you will implement encryption and decryption for substitution ciphers. As mentioned

mentioned, we will use the Key type, which is an association list for Char to Char substitutions. The elements

in the list tell which characters to substitute. For example, if the list contains

the pair ('a','x'), all occurrences of 'a' in the input should be replaced with 'x'.

ATTENTION: If a character does not exist in the key, that character should be retained.

For both plain text and ciphertext, we should use the String type. Remember that String = [Char].

a) Write a function encode :: Key -> String -> String that given a secret key

encrypts a plaintext. Hint: If you use lookup you can use a case-split or

fromMaybe to retain characters that are not in the key.

b) Write a function decode:: Key -> String -> String that given a secret key

decrypts a ciphertext back to plaintext. Hint: The actual code of decode will be very similar

very similar to the code of encode. If you define a helper function invert :: Key -> Key that

reverses the key, you can define decode using encode.

c) Write a function caesar :: Alphabet -> Integer -> Key that creates a reshuffle key

that rotates the alphabet a given number of steps. You can use this to

test encode and decode.

As a simple test of your functions, you can load the program into GHCi and run the following:

> caesar "abcde" 2

[('a','c'),('b','d'),('c','e'),('d','a'),('e','b')]

3

> alphabet = ['a'..'z'] ++ ['A'..'Z'] ++ " .-?!"

> key = caesar ['a'..'z'] 5 ++ caesar ['A'..'Z'] 5 ++ caesar " .,-?!" 5

> plaintext0 = "Hello, world!"

> ciphertext0 = encode key plaintext0

> putStrLn ciphertext0

Mjqqt.!btwqi?

> decode key ciphertext0 == plaintext0

True

> plaintext1 = "Symbols: # $ %"

> ciphertext1 = encode key plaintext1

> putStrLn ciphertext1

Xdrgtqx:!#!$!%

> decode key ciphertext1 == plaintext1

True

Task 2: Frequency tables

The reason why substitution ciphers are so vulnerable is that every occurrence of a character in the plaintext is encrypted

to the same character. In our language, different characters occur with different frequencies. For example

space will appear very often, as it is used between every word in every sentence. Other

characters, such as the letter "x", are not used as often. You can therefore look at the ciphertext to see which characters

that occur frequently and start guessing which characters they correspond to in the plaintext.

The first thing we need is a function that calculates the frequency of each word in a text. The frequency

is defined as the number of occurrences in a text, divided by the length of the text. We use double precision floats

to represent the frequency, and the typeFrequencyTable that associates each character

with its frequency.

Such a frequency table is a simple model of the language. We will use the model to try to recognize

recognize when we are approaching the correct key by comparing the character frequency with the model. This

This comparison is called "chi squared"

a) Write a function count :: String -> FrequencyTable that calculates the frequency for each

character in a string. Hint: Try to avoid calculating the length of the list multiple times. The naive

algorithm for counting the number of occurrences will have quadratic time complexity, as it iterates

through the entire rest of the list for occurrences. There are simple solutions that are more

efficient, for example by sorting, or going via Data.Map. These will instead have n-log n

time complexity.

To easily create a frequency table from a corpus, we need a function that reads in a file

and apply count to the contents of the file.

b) Write a function loadFrequencyTable :: FilePath -> IO FrequencyTable that reads in

a file and creates a frequency table for the contents. Hint: The function readFile :: FilePath

-> IO String reads in an entire file.

The next step to crack the code is to make a simple guess at what the secret key might

based on the model and the observed frequency. This guess is not necessarily

correct, but will serve as a starting point for further improvement.

c) Write a function initialGuess :: FrequencyTable -> FrequencyTable->Key that

matches two frequency tables by associating the character with the highest frequency in one table

with the one with the highest frequency in the other, and so on downwards. If one of the tables

is shorter than the other, the association stops when the shorter table ends. Hint: Use the

the sortBy function to sort the tables correctly. After sorting, you can either

project away the frequencies and use zip, or combine with zipWith.

Now we will calculate the x^2 ‘’chi squared ‘’ statistic. It measures how close an observed frequency table is to the model.

The mathematical formula for x^2 ‘’chi squared ‘’



where S is the set of all possible outcomes

(in our case, S is all characters that occur in the model or string). In this formula, Oi

stands for the observed frequency and Ei the expected frequency according to the model.

An additional complication is that we may observe a character that does not occur in the model.

In that case, the formula above will contain division by zero. A standard technique to handle this is to

assign a low but non-zero probability to such signs. This is called a smoothing factor:

smoothing factor). You can use a smoothing factor of 1/10000.

d) Write the function chiSquared :: FrequencyTable -> FrequencyTable -> Double that

calculates the 2 statistics. The first argument is the model, i.e. the expected frequency table.

The second argument is the observed frequency. Hint: The textbook has an example

implementation of 2 for a slightly simpler case. You can use it as a starting point, but

you need to take into account that we are working with association lists here. One approach would be to create

a list of all unique characters, and then look up each character in both frequency tables (and use the

smoothing factor if needed) to calculate each summand.

As a simple test of your capabilities, you can load the program into GHCi and run the following:

> count "Hello"

[('H',0.2),('e',0.2),('l',0.4),('o',0.2)]

> import Data.Tuple (swap)

> import Data.List (sortBy)

> model <- loadFrequencyTable "corpus.txt"

> -- Find the number of unique symbols in the corpus

> length model

87

> List the symbols of the corpus (order might vary)

> map fst model

"\n\r !\"#$%&'()\*+,-./0123456789:;<=>?

ABCDEFGHIJKLMNOPQRSTUVWXYZ]abcdefghijklmnopqrstuvwxyz"

> Define a natrual sorting for frequency tables

> oplex x y = compare (swap y) (swap x)

> List of symbols in the corpus, sorted by frequency

> map fst $ sortBy oplex $ model

" etaoinsrhldcumpfgywb.,vkIT-A'S1\"0Cx

)(2M9OPEBWNRHDLFjG3z;q5:8476U?J#YK/V&\*!%=<>$\r\nQZX+]"

> plaintext2 = "Values are becoming more and more

recognized as issues in psychotherapeutic discourse"

-- Unique characters in the plaintext, sorted by frequency

5

> map fst $ sortBy oplex $ count plaintext2

"e sroicaunmdtphgzylbV"

> key = caesar ['a'..'z'] 5 ++ caesar ['A'..'Z'] 5 ++ caesar " .,-?!" 5

> ciphertext2 = encode key plaintext2

> map fst $ sortBy oplex $ count ciphertext2

"j!xwtnhfzsriyumlqgedA"

> guessedKey2 = initialGuess (count plaintext2) (count ciphertext2)

> decode guessedKey2 (encode key "home") == "home"

True

> Test that the plaintext is a better fit for the model than the ciphertext.

> chiSquared model (count plaintext2) < chiSquared model (count ciphertext2)

True -- In fact, chiSquared model (count plaintext2) should be a bit less than 1

-- while chiSquared model (count ciphertext2) should be more than 100

Task 3: Greedy optimization

In this task, we will use a greedy algorithm to improve the guess of what the secret key

key was, given only the ciphertext as input. The idea is simple: a correctly decrypted text will

have frequencies similar to those of natural language. Given a frequency table that models natural language natural language, we can use x^2 to measure how close the decrypted text comes to the model: Lower x^2 is a better fit to the model. In each step of the algorithm, we try all possible substitutions of key values and, greedily, take the one that minimizes x^2. This continues until no swaps yield a x^2.

Caveats: The above algorithm is not very fast and does not necessarily yield the key with the x^2.

Greedy algorithms can get stuck in a local minimum. It is also far from certain that the

secret key is the one that minimizes x^2. We can therefore not expect a perfect decryption

from this algorithm. In the next task, we will take a further step to improve the guess on

what the secret key might be.

We implement the algorithm in a couple of steps.

a) Write a function, swapEntries :: (Char,Char) -> (Char, Char) -> Key -> Key, that

swaps two of the subscripts in a key. Hint: You can use map with a helper function

to go through all the key's values and update if it matches one of the pairs to be

to be swapped.

The function swapEntries can be written to work on all association lists.

b) Change the type of swapEntries so that it is as general as possible.

In the greedy algorithm, we need a list of immediate changes that can be made to the key,

so we can figure out which one is best.

c) Create a function, neighborKeys :: Key -> [Key] that given a key generates all keys

you can get from it by swapping two different elements.Hint: You can use list comprehension

together with swapEntries to solve this task.

Now we're going to put together the functions we've created so far to implement a greedy algorithm

that attempts to find a key that minimizes x^2 of the decrypted text relative to a given

model.

d) Write a function greedy :: FrequencyTable -> String -> Key -> Key that given a model

and a ciphertext, and an initial guess on key, greedily guesses a key

that decrypts the ciphertext so that x^2 relative to the model is minimized. Hint: You can

use recursion to implement the greedy algorithm. You can use map or list comprehension

to go through the neighborKeys and use decode and calculate the chiSquared for

each of the new keys. Then you can use minimum to find the new key that has

at least x^2 and compare with the given key and its x^2.

As a simple test of your functions, you can load the program in GHCi and run the following:

> swapEntries('a','b') ('b','c') (caesar "abcde" 1)

> Test the generalization of swapEntries by

> -- applying it to integers and functions:

> swapEntries (1,2) (2,1) [(1,2),(2,1)]

[(1,1),(2,2)]

> ($ 0) <$> lookup 1 (swapEntries (1,cos) (2,sin) (zip [1..3] [cos,sin,tan]))

Just 0.0

> Test the neighbourKeys function (order of elements may vary)

> neighbourKeys (caesar "abc" 1)

[[[('a','c'),('b','b'),('c','a')]

,[('a','a'),('b','c'),('c','b')]

,[('a','b'),('b','a'),('c','c')]]

> model <- loadFrequencyTable "corpus.txt"

> plainText3 <- take 2000 <$> readFile "corpus.txt"

> key = caesar ['a'..'z'] 3 ++ caesar ['A'..'Z'] 3 ++ caesar " .-?!" 3

> cipherText3 = ekncode key plainText3

> 0 = initialGuess model (count cipherText3)

> kg = greedy model cipherText3 k0

> Test that 2 is small:

> chiSquared model (count (decode kg cipherText3)) < 0.1

True

Task 4: Dictionary optimization

If you try to decrypt using the key as the greedy algorithm in the previous task

you will most likely not get anything useful out of it. The best we can hope for is that the most common

symbols, such as spaces and the letter "e" are in place. For example, compare the beginning

of the corpus with the attempted decrypted string:

"It is safe to say that ours is the only dining room in West Los Angeles"

"ka is stpe ao stu ahta ogrs is ahe onlu dininw room in Hesa 0os Snweles"

Not exactly a perfect match, but some key symbols are in place. Could a few changes

will result in a perfect match? The next idea is to do another greedy algorithm, but this time we try to

we try to make the text consist as much as possible of words from the language we are decrypting into. In our case English.

We start by creating a dictionary ourselves, from a corpus located in a file. The type Dictionary

type represents dictionaries as a Data.Set of strings.

a) Write a function, loadDictionary :: FilePath -> IO Dictionary, that reads a corpus

and finds all unique words that occur. Hint: Once the file is loaded, you can use the functions

words and Set.fromList functions to create the dictionary.

The next step is to use a dictionary to count how many valid words occur in a string.

b) Write a function, countValidWords:: Dictionary -> String -> Integer that counts

how many of the words in a string occur in the dictionary. Hint: You can use filter

together with Set.member to filter the words that occur in the dictionary.

Finally, we will implement the new greedy algorithm. It will try to maximize the number of

valid words in the decrypted text. Or is the setup the same as in the previous task. We begin

with an initial guess at the key, and a dictionary. Then we try all simple substitutions of values in the key

key and see if they improve the number of decrypted words in the dictionary. We continue in this way until

no swaps result in more dictionary words.

c) Write a function, greedyDict:: Dictionary -> String -> Key -> Key, that implements

the greedy algorithm to find the key that maximizes the number of dictionary words in the

the decrypted text. Hint: You can use the same setup as in greedy, with recursion and

using the neighborKeys functions, but instead of chiSquared and minimum, you must use

countValidWords and maximum.

As a simple test of your functions, you can load the program in GHCi and run the following:

> dictionary <- loadDictionary "corpus4.txt"

> Set.member "the" dictionary

True

> countValidWords dictionary "hello world"

2>

countValidWords dictionary "not quite carrect"

2>

plainText4 <- take 2000 <$> readFile "corpus4.txt"

> model <- loadFrequencyTable "corpus4.txt"

> key = caesar ['a'..'z'] 3 ++ caesar ['A'..'Z'] 3 ++ caesar " .-?!" 3

> cipherText4 = encode key plainText4

> k0 = initialGuess model (count cipherText4)

> kg = greedy model cipherText4 k0

> kd = greedyDict dictionary cipherText4 kg

> A simple function to test the percentage two strings match

> match a b p = 100\*(length $ filter id $ zipWith (==) a b) > p\*(length a)

> Check that decrypted cipher text matches plain text at least 95%.

> match (decode kd cipherText4) plainText4 95

True