Vodro :

Hello good people. He is Robin, an NLP enthusiast. As we all know, NLP tasks require tons of string text. For training a new model he needs more data in his local machine. But all his drives are full. So he needs to buy a new hard drive. But due to corona pandemic, moreover the shutdown, it is hard to manage a new hard drive. So he started to ask,

-> Can we store more data in the same disk space?
I, Kowshic Roy, with my partner Mr. Mahdi Hasnat Siyam, will go deep down to answer this question in this presentation.

We all know about ASCII , where each character is assigned a 8 bit unique code. ASCII is an example of fixed length encoding where each character has the same length code.

For encoding a text like “JAVA” with ASCII we take the first character “J” and put its ASCII code, Similarly we do the same for next characters A , V and again A.

To decode the message, each time we take 8 bits and write their corresponding ASCII character. Thus we regain our encoded text “JAVA”.

Suppose our string of interest is ABRACADABRA the magical word.
In ASCII representation we need a total of 120 bits.

Let’s explore how we can reduce the bits count.

One thing is to notice here is that we are always assigning 8 bits to each character.
But why are we always taking 8 bits?
Here we have only 5 unique characters. So this can be uniquely represented by 3 bits.
So we make our own encoding table assigning 3 bits to each character. As we can see, we need a total of 45 bits for representing our string.

Mahdi: But, How will the decoder know about this encoding?

//Suppose you are the decoder. How will you know about this encoding?
Vodro: You are right. We also need to save our encoding table with the encoded message. But this is of almost constant size with respect to the size of the text stream. So we can safely ignore the bits needed for the encoding table for the rest of our presentation.

So we saw, intelligent encoding can save space.

If you look closely, doesn't it feel that the leading zeroes are redundant? Why don't we throw them out?
Now our modified table is like this.
For the time being please trust me somehow this encoding works.
Here we need a total of 36 bits.
This is our first variable length encoding table.

Mahdi: Why are we not using the frequency of characters? Wouldn't it be optimal to assign small length code to higher frequent characters.

//One observation is that, why are we not using the frequency of characters.
Vodro: Yes, you are absolutely right. We sort our table with respect to frequency of characters so the highest frequent character has the lowest possible length code.
This time we need a total of 20 bits.
So, frequency based encoding is a good approach in reducing bits count.
//But what is the use of encoding if it can't be decoded rightly.
Can our encoding be decoded properly?
Suppose we have an encoded message 10. So what is the main text? Is it "BA" or "R" !

So something is wrong with our table.
Looks like we are stuck here. Mr. Mahdi, can you guide us to the correct variable length encoding scheme?

Mahdi : Yes I will be pleased to help.

In What went wrong

Here one thing we can notice from the table that 1 is a prefix of 10.
So when we get 1 we cant decide if it is a part of R or the B itself.

➡
In Properties of correct encoding -> prefix property
[28]
Therefore for our encoding table to work properly,
Property it should have is that no whole code word is a prefix of other code word
➡
Encoding in the left table satisfies prefix properties while the right one doesnt.
Because in the right table 0 is prefix of 01 .
➡

In encoding as trie -> Representation of encoding as trie

[30]
➡
[31]
Now keeping prefix property in mind ,
➡ we create a trie consisting of code.
Here left child denotes 0 and right child denotes 1.

vodro: So what about prefix property in trie encoding?

Mahdi: yeah , prefix rule is satisfied if and only if every character is stored in external/ node

➡
In encoding as trie -> decoding from trie

Now we have a trie that represents encoding , and encoded bits,
Lets see how we decode text using this encoding trie.

For that we will maintain a pointer to the trie .
Initially it is set to root node,
Now our next bit is 1 so we move to right child.
Next we reach to leaf node , here we write corresponding character to decoded text , and move our pointer to root node , similarly we can decode the whole text Java.

So using trie we can decode text in linear time

➡
In optimal compression -> example 1
[33]
➡
[34]

For our text with this encoding trie total encoded bit length is 41 .

➡
[35]

Again in this example we Reordered some characters in the trie .
We assigned less length code for frequent character which we have seen already. Now we only need 32 bit to encode same text.

➡
In optimal compression -> construction

So total bit length depends structure of the encoding trie , we want to adopt a systematic approach to find encoding trie that minimize total bit length.

➡
We will use frequency property.
We will build the trie in bottom up manner.
Initially in a list we have every leaf node , associated with character and having weight equal to frequency of that character.
➡
At every step we choose two node having least weight.we will Create new node combining them and delete these nodes from the list.
We will add the new node to the list

➡
Here we have three nodes with least equal weight.
In this case we can proceed with any two of them.
Now taking node with weight 4 and 2
Here we get our Final huffman encoding trie.
Again if we calculate total bit length it is 27 which is the minimum possible.
The process we followed is called huffman coding.

=== Result====
Vodro: As we can see with respect to storage consumption, our huffman encoding performs 4 times better than standard ASCII and roughly 2 times better than our designed fixed length encoding.
I hope this makes Robin happy.
Thank you all for your kind attention.

==
//Given this encoding table and this encoded message, we recursively try to match with the codes and write corresponding characters.
//One interesting thing with this encoding table, is that we can take either 0 or 01. So we have an ambiguous decoding.
//The reason behind such ambiguity is that, code 0 is a prefix of code 01.
The property a encoding table should always have is , no whole code is a prefix of other code words. The left encoding satisfies the prefix property while the right table doesn't.
//Mr. Mahdi can you lead us to a correct variable length encoding scheme?

In section Decoding Scheme for var length code
[25]
Mahdi: Now the problem is, how do we decode?
➡

In Example -> Correct Decoding Slide

[26]

Mahdi :lets say someone gave us an encoding , and an encoded text.
lets see how we decode the coded bit stream to original text.
To do that, we recursively find matching code starting with our encoded text and replace it with corresponding character.
➡
Here we found J ➡ then A ➡ then V➡and again A
➡
In Example -> Ambiguous Decoding
[27]

Here we have a different encoding and encoded text.
Lets try it
➡Here again ➡we found J ➡ then ➡ A ➡now we have two matching character , we can either replace 01 with V or 0 with A ➡ then we continue our matching
➡➡ Here ➡ we get JAVA and
4 * ➡
So we dont know which text was encoded originally.
So main problem is when wo found 01 we ware not sure which code to take 0 or 01.
➡
There's an important fact that 0 is prefix of 01

//First 8 bits correspond to J but next we have 6 bits which is less than 8. It is an
//example of failed decoding.