Chapter 4

**Project: ChatBot**

**Data Preprocessing**

**4.1 Import Libraries**

There are only four libraries we need to use for data-preprocessing.

# *Building a ChatBot with Deep-NLP*

**import** numpy **as** np

**import** tensorflow **as** tf

**import** re

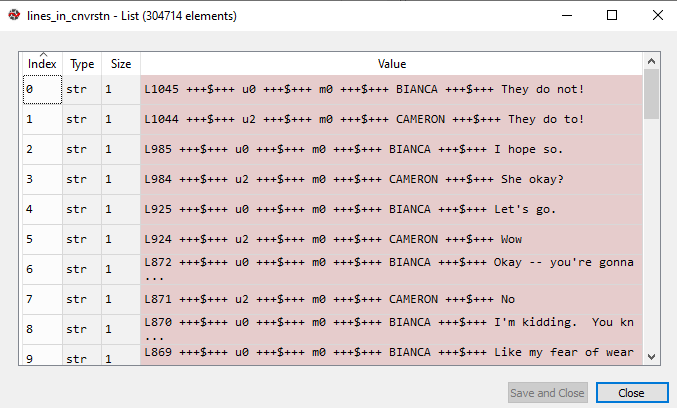
**import** time

1. ***Numpy*** to work with arrays.
2. ***Tensorflow*** to build Deep-NN
3. ***RE*** library which will use to clean the text and make them as simple as possible. Also used to replace some characters by some more simple characters. To simplify the conversations as much as possible.
4. ***time*** library to measure the training time of each epoch. To keep track of the training.

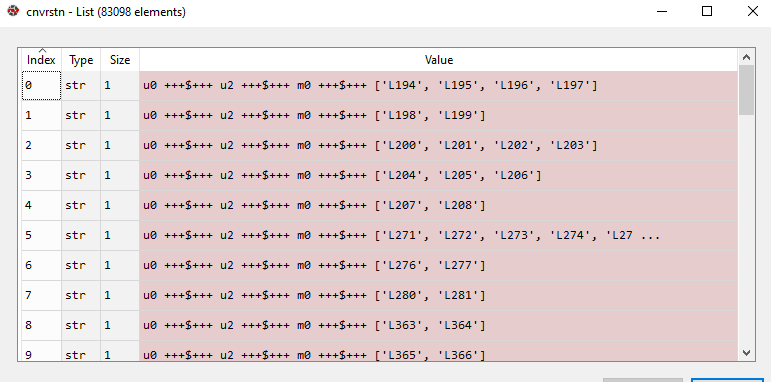
**4.2 Import the DATASET**

We’ll import two dataset from the downloaded data. "movie\_lines.txt" and "movie\_conversations.txt" I two separate variables.

* Following is the data of the movie lines:



* Following is the data of the movie conversations:



* We can see each movie lines has an unique identifier id, those ids of the lines are used in the conversation dataset.
* Now we are going to create a ***dictionary***. This dictionary will ***map*** each ***lines-ID*** with its text.
* We already have a *mapping* between the *key-Identifiers* and the *lines* in conversation dataset. But we want to make a *python* *dictionary* because we will use it afterward to get our *data* ready for the *future NN* that we're going to *build*.

**4.3 Creating a Dictionary**

We want our dataset composed with two columns, the *input* and the *output*. Because the *inputs* will be fed into the *NN* and the *outputs* will be the *target*.

* We will compare the target replied by real people with the ChatBot. To know how the ChatBot will learn to speak whether its prediction is close to the target.
* The easiest way is to create a dictionary. Because we'll have to keep track of the conversations to make a correct mapping between the inputs and the output.
* Remember, you'll have to do a lot of cleaning in your dataset in order for the training to go smoothly. They need to be simplified as much as possible. This dictionary is the part of this cleaning process.
* The *key identifier* of the dictionary will be the *ID* of the *line* and the *value* will be the *line itself*.
* We are going to iterate through all the lines of our dataset. Each of these lines we are going to split and they are separated by **+++$+++**.
* We will get the key-ID which is the first element and then we will get the value that is the line-itself which is the last element.
* Note: add space " **+++$+++** " because if you look *closer* there is actually some spaces between the *elements* and this string **+++$+++**.

id\_2\_line = {};

for lyn in lines\_in\_cnvrstn:

    \_line = lyn.split(" +++$+++ ") *#returns a list*

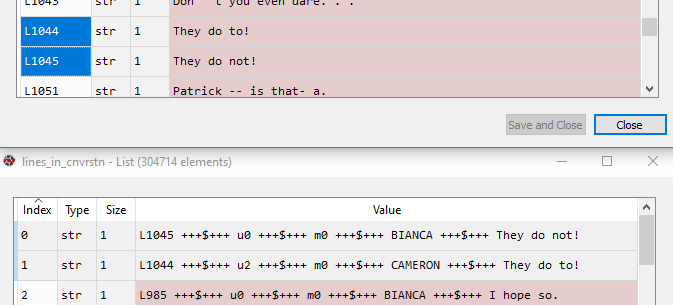
* That makes the split and then we're going to get *the line ID* and the *line itself*.
* **split()** Splits the string at the specified separator, and returns a list.
* For safety purpose we're going to make sure that the line has the correct length which should be 5. Because 5 elements are separated by " **+++$+++** "

**if** **len**(\_line) **==** 5:

        id\_2\_line[\_line[0]] = \_line[4] # *creates the dicttionary*

        # *\_line[0] is id "key in dictionary" and \_line[4] is the line "as value of the key"*

* **len**(\_line) **==** 5 Checks if the list formed by **split()** has 5 elements.
* id\_2\_line[\_line[0]] = \_line[4] put the key-value pair in our empty dictionary.
* \_line[0] is id "as the key in dictionary" and \_line[4] is the line "as value of the key"



**The complete code to create dictionary**

# *Creating a dictionary that maps each linea and its ID*

id\_2\_line = {};

**for** lyn **in** lines\_in\_cnvrstn:

    \_line = **lyn.split**(" +++$+++ ")

**if** **len**(\_line) **==** 5:

        id\_2\_line[\_line[0]] = \_line[4] # *creates the dictionary*

        # *\_line[0] is id "key in dictionary" and \_line[4] is the line "as value of the key"*

* We got our dictionary now. Now we're ready to move onto the next step which is to create a list of all the conversations with the IDs as we got in the "movie\_conversations.txt" data-set.
* In training, we need to *keep track of the conversations* for the training data.

**4.4 Creating a List-of-Conversation**

We're going to create a list of all the conversations. (i.e. it will be list of list-of-lines, *list-of-lines* creates the *conversation*).

* We already have a *list* of the *conversations* but we have a lot of *metadata* and we want to only use the data which are useful for *training*. From this dataset we just need *the "list-of-lines"* that makes the *conversation*.
* We'll introduce a new empty list called **cnvrstn\_ids = []**
* We use for loop to loop through "movie\_conversations.txt" data-set.
* We first **split()** wrt **" +++$+++ "**
* Then we **select** the **last element** of the list formed by **split()**, which is list of **ids**.
* Then we eliminate, "**[**" and quote-sign from the **ids**.
* The *last row* of the *conversations* *data* *set* is an *empty* *row* so we won't include it and therefore we'll use a **-1** because it includes all rows except the last row.

# *Creating the list of all of the conversations*

cnvrstn\_ids = []

**for** cvstn **in** cnvrstn[:-1]:

* First step: \_cnvrstn =  **cvstn.split**(" +++$+++ ")[-1][1:-1].**replace**("'", "").**replace**(" ", "")
* **cvstn.split**(" +++$+++ ") returns the list formed by **split**(" +++$+++ "),
* **cvstn.split**(" +++$+++ ")[-1] returns last element of the list formed by **split**(" +++$+++ "), ie the list-of-ids like ['L194', 'L195', 'L196', 'L197'].
* But remember "['L194', 'L195', 'L196', 'L197']" is actually a ***string***, returned by ***split()***,
* **cvstn.split**(" +++$+++ ")[-1][1:-1] excludes the "[" and "]" from the string of id-list:
* in [1:-1], 1 specifies string starting from index 1, excluding index-0 and -1 specifies to exclude the last index.
* **replace**("'", "").**replace**(" ", "") finally replaces ' and spaces by nothing. Finally we get a string like:

"**L194,L195,L196,L197**"

* Second step: During appending the id-lists to our conversation-list ***cnvrstn\_ids***, we once again apply ***split()*** to the id-list-strings "L194,L195,L196,L197", by using **\_cnvrstn.split**(","). Since "L194,L195,L196,L197" is stored in **\_cnvrstn**.

**cnvrstn\_ids.append**(**\_cnvrstn.split**(","))

**The complete code to create List-of-conversation**

# *Creating the list of all of the conversations*

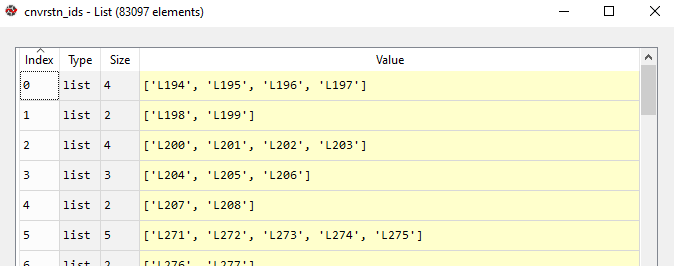
cnvrstn\_ids = []

**for** cvstn **in** cnvrstn[:-1]:

    \_cnvrstn =  **cvstn.split**(" +++$+++ ")[-1][1:-1].**replace**("'", "").**replace**(" ", "")

**cnvrstn\_ids.append**(**\_cnvrstn.split**(","))

* finally we got following kind of list/dataset:

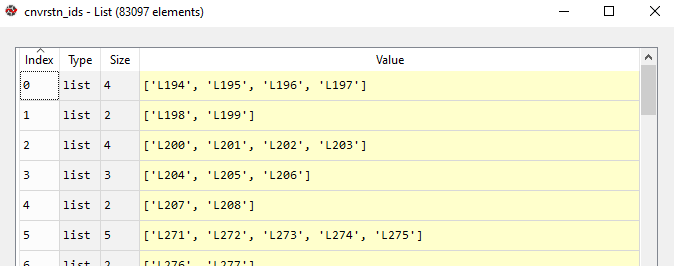


Next we sort the lines into some questions, which will be the ***input*** of the ***NN*** and the answers which will be the ***targets*** of the ***NN***.

**4.6 Separating Questions and Answers**

We want to get two separate huge lists. One for the questions and one for the answers but of the same size because for each index i of this list, the answer of index i should be the question of index i, therefore it should be well aligned.

* In our ***conversation Dataset***, notice all line-ids are in ***pair***, the first-one (eg: ***L194***) is a question, and the second one (***L195***) is answer. Then in next second one (***L195***) becomes the question for the third-line (***L196***), so ***L196*** is ans to ***L195***, and so on.



# *Getting seperately the questions and answers*

questn = []

ans = []

**for** cvstn **in** cnvrstn\_ids:

**for** i **in** **range**(len(cvstn) - 1):

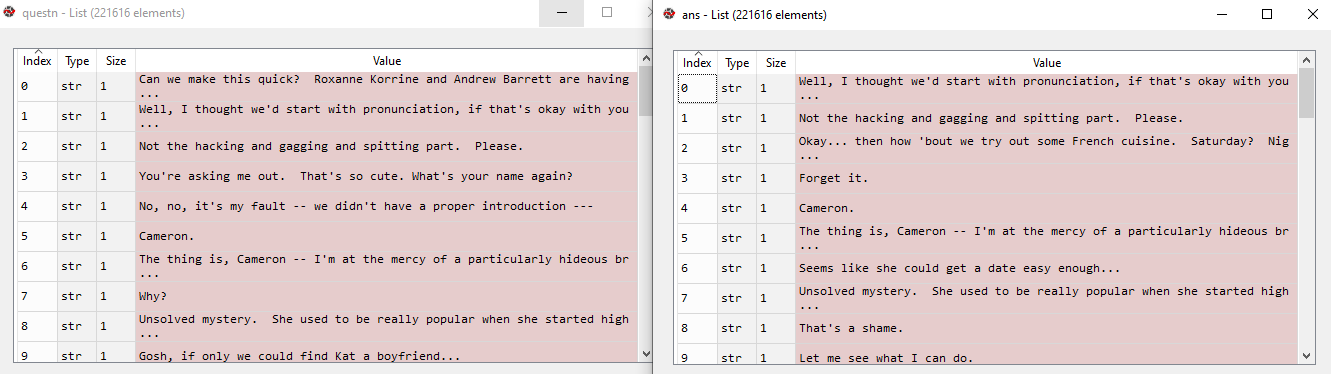
**questn.append**(id\_2\_line[cvstn[i]])  # *using the "id\_2\_line" dictionary, by id-key*

**ans.append**(id\_2\_line[cvstn[i+1]])

        # *range(len(cvstn) - 1) is used because of cvstn[i+1]*

        # *notice we are using both "cnvrstn\_ids" and "id\_2\_line"*

* We then get following:



**4.7 Cleaning**

We put everything in *lower-case*,

We'll remove all of the *apostophies*, eg: "that's" same to "that is"

We'll remove all *non important* words

* We'll make a function "***clean\_text(text)***", which applied on a text and do multiple cleaning at the same time.

# Doing the first cleaning of the texts

def **clean\_text**(text):

    text = text.lower();

    text = re.sub(r"I'm", "I am", text)

    text = re.sub(r"he's", "he is", text)

    text = re.sub(r"she's", "she is", text)

    text = re.sub(r"that's", "that is", text)

    text = re.sub(r"what's", "what is", text)

    text = re.sub(r"where's", "where is", text)

    text = re.sub(r"\'ll", " will", text)

    text = re.sub(r"\'ve", " have", text)

    text = re.sub(r"\'re", " are", text)

    text = re.sub(r"\'d", " would", text)

    text = re.sub(r"won't", "will not", text)

    text = re.sub(r"can't", "cannot", text)

    text = re.sub(r"[-()#/@;:<>{}+=~|.?,]", "", text)

    return text

* Be careful "-" and "~", problem appeared.
* We can add more replacement to this function.
* Cleaning the questions & answers: We'll make two new lists of cleaned questions and cleaned answers.

# Cleaning the questions

clean\_questn = []

for qes in raw\_questn:

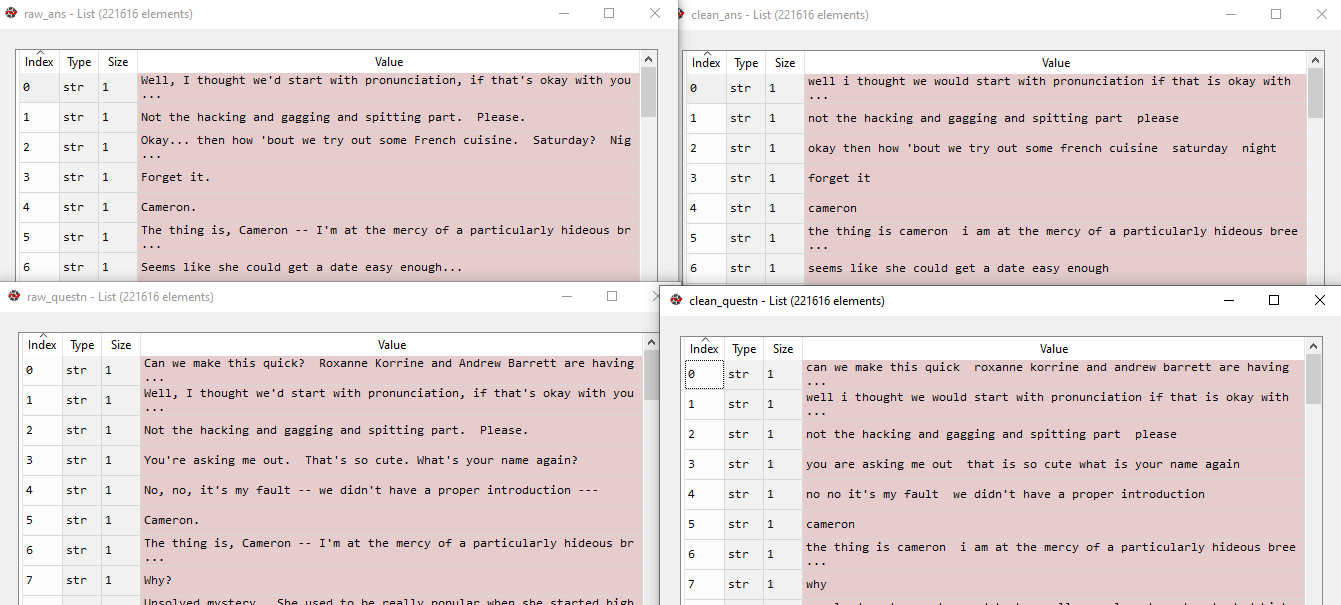
    clean\_questn.append(clean\_text(qes))

# Cleaning the answers

clean\_ans = []

for aNs in raw\_ans:

    clean\_ans.append(clean\_text(aNs))



* Removing No-frequent words: We'll create a dictionary that maps each word to its number of occurrences.
* This dictionary will be very useful for us to ***remove the words*** that are ***below*** a certain ***threshold of frequency***. After creating this dictionary, we will choose this threshold and then count the number of words that appear more than the chosen threshold.

# *create a dictionary that maps each word to its number of occurrences.*

word2count = {}

**for** questn **in** clean\_questn:

**for** word **in** **questn.split**():

**if** word **not** **in** word2count:

            word2count[word] = 1

**else**:

            word2count[word] += 1

**for** ans **in** clean\_ans:

**for** word **in** **ans.split**():

**if** word **not** **in** word2count:

            word2count[word] = 1

**else**:

            word2count[word] += 1

**4.8 Tokenization and Filtering**

* Tokenization and Filtering: Now we create two dictionaries that map the *questions words* and the *answers word* to a *unique integers.*
* While creating those dictionaries we'll do two things:

1. Tokenization
2. Filtering non-frequent word

* We will add an *if condition* that will *check* if the *number* of *occurrence* of the *word*.
* If the number is ***higher*** than a ***certain threshold*** we will ***include*** that word in the dictionary, this word will be one of the ***key*** ***identifier*** of the dictionary.
* If the number of *occurrence* of the word is below the threshold, we will not include the word in the dictionary.
* So at the same time we proceed
* Tokenization by getting separately all the *different words* from the *questions* and the *answers* and attributing a *unique* *integer* to each of this word.
* Filter out the words that don't appear frequently in the dictionary
* Threshold: So at first we will set this threshold number:
* Generally we can filter-out ***5%*** of the whole corpus, however according to size of our corpus of word, to remove ***5%*** we use the value between ***10*** and ***20***.
* We can tune this value as a Hyper-parameter.
* We choose ***20***, this will remove ***5%*** of the word from our corpus.
* For faster training you can choose higher values.
* If a powerful computer available, we can use lower value.
* Remember, too low or too high values can result overfitting/underfitting, so be careful.

# *Creating two dictionaries that map the questions words and the answers word to a unique integers*

threshold = 20

word\_number = 0

questnWrd2Int = {}

# *'word' is "key" and 'count' is "value"*

**for** word, count **in** **word2count.items**():

**if** count **>=** threshold:

        questnWrd2Int[word] = word\_number

        word\_number += 1

* That creates our questions word to number dictionary (maps each most frequent word to a unique integer). So we get this dictionary that encodes each of the words of the questions at the same time only including the frequent ones. Or at least the 95% most frequent words and now we're going to do the same for the answers

word\_number = 0

ansWrd2Int = {}

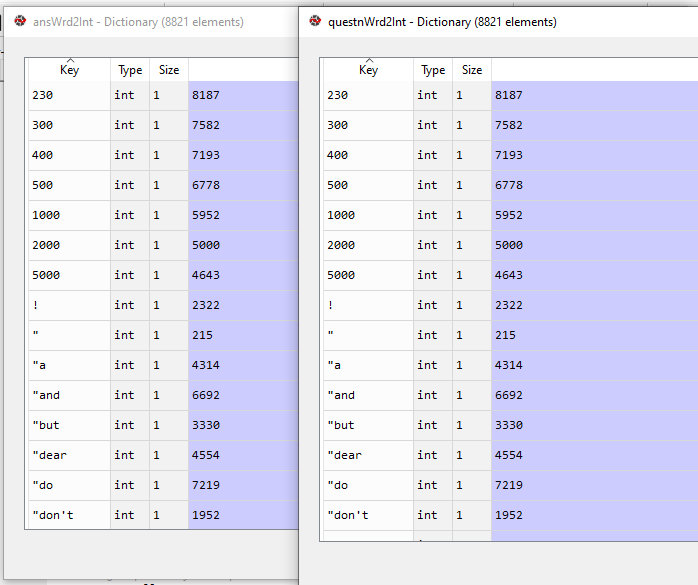
**for** word, count **in** **word2count.items**():

**if** count **>=** threshold:

        ansWrd2Int[word] = word\_number

        word\_number += 1

* Notice that, the two dictionaries ***questnWrd2Int*** , ***ansWrd2Int*** are the same. We do it just for the training purpose.



* The next step will be to add the final unique tokens to our two dictionaries that we just created. And that's basically the ***END OF STRINGS (EOS)***, ***START OF STRING (SOS)***, which are essential unique-tokens to building the Seq2Seq network.
* SOS also called <GO>
* Adding SOS, EOS and PAD:
* **PAD: PAD** (Padding) which is very important for our model because the process turning *data* and the *sequences* in the *batches* should all have the *same length* and therefore we have to *input this token* in an *empty position*.
* **OUT:** we will create a last token which will called ***OUT*** and that *corresponds* to all the *words* that were *filtered* *out* by our two *previous dictionaries*.
* We know which words are not the 5% least-frequent words. Later we'll use these dictionaries to replace all these 5% least-frequent words. These will be replaced by only one token **OUT**.
* In following code, the order of the tokens must be followed:

# Adding the last tokens to these two dictionaries

tokens = ['<PAD>', '<EOS>', '<OUT>', '<SOS>']

for tkn in tokens:

    # adding 'token' as "key" and 'unique int' as "value"

    questnWrd2Int[tkn] = len(questnWrd2Int) + 1

for tkn in tokens:

    # adding 'token' as "key" and 'unique int' as "value"

    ansWrd2Int[tkn] = len(ansWrd2Int) + 1

We now have our final tokens the essential tokens for the Seq2Seq added to our two dictionaries.

* The next step will be to create the Inverse Dictionary Of The Answers words to in dictionary. Because we'll use this inverse mapping when building the Seq2Seq model.

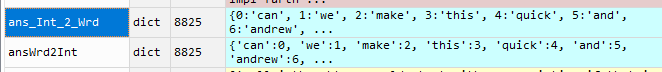
**4.9 create the Inverse Dictionary**

We will need *inverse mapping* from the *integers* to the *answers* *words* in the implementation of the Seq2Seq model. Currently we have mapping for *answers* *words* to *integers*.

* We actually need it for the answers-words dictionary and not the questions-word.
* Inversing dictionary in Python is easy; we just need to swap the keyword and corresponding value:

# Creating inverse dictionary of the ansWrd2Int, notice ':' is used to crete dictionary

ans\_Int\_2\_Wrd = {w\_i: w for w, w\_i in ansWrd2Int.items()}



* Next step: We need to add EOS to the end of every answer. That's for the *decoding* part of the Seq2Seq model. We'll make a quick for-loop for that.

Then we'll move on to the last two big steps of this data processing part of NLP that will considerably improve the training process.

**4.10 Adding EOS tokens to the end of every answers**

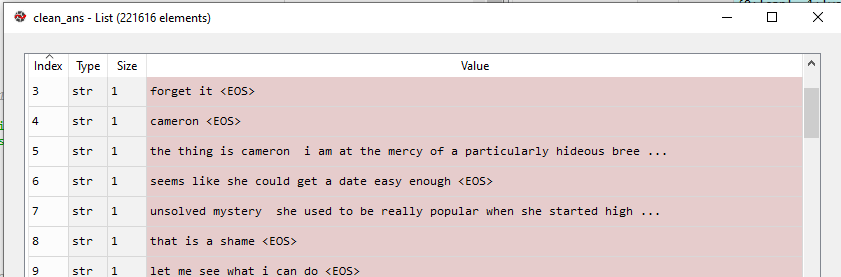
We're going to ***loop over*** all the answers in our ***clean answers***. And we add EOS one by one.

Well since we already have the clean, we can directly start the loop.

# *Adding EOS tokens to the end of every answers*

**for** i **in** **range**(len(clean\_ans)):

    clean\_ans[i] += " <EOS>" # *notice a space is added to seperate <EOS>*



* Be careful to make that *separation between* the *last word* and the *answer* and this **<EOS>** token, it is necessary to add the space here so that it can clearly distinguish the last word and the **<EOS>** token. Otherwise it would confuse this.
* Next two steps will be easy, these are the last steps of data preprocessing.
* The first one will be to remove all the non frequent words. That is the 5% ***least frequent words*** from all the ***questions*** and the ***clean questions*** ***list*** and also the ***answers*** and the ***clean answers list***.
* Lastly we'll *convert* all of these *cleaned* and *filtered* Questions and Answers to ***integers*** that will be our last conversion.

Because in the end we will sort the questions and answers by the ***length of the questions***. to improve training-process and make it shorter and optimized.

**4.11 Converting sentences to "sentence of integers" using previously created TOKENIZED dictionaries**

Here we take our clean questions: that are into text and we want to translate that into the unique integers that were mapped to our previous dictionary that we created.

* We're only doing this because we want to *sort* all the *questions* and *answers* by their *length*. We want to sort the questions and answers by the length to *optimize* the *training performance*.
* Let's make our two lists of questions and answers translated into **integers** and also while making that list we're going to replace all the words that were filtered out by our token **<OUT>**.
* So at some point in the loop we will need to add an ***if condition*** to check if the word was filtered out and if that's the case will replace this word by the token **<OUT>**.
* We have to do two four loops. First we need to loop over all the questions. then the second for loop to loop over all the words of the questions.
* Here we'll use our previous tokenized dictionaries, ***questnWrd2Int*** and ***ansWrd2Int*.** And remember these are the dictionaries those map each word of the question/answers to an integer.

# *Translating all the Questions and the Answers into integers*

# *and replacing all the words that were filtered out by our token <OUT>*

questions\_to\_int = []

**for** question **in** clean\_questn:

    ints = []

**for** word **in** **question.split**():

**if** word **not** **in** questnWrd2Int:

            # *Checking filtered word and adding <OUT>'s token*

**ints.append**(questnWrd2Int["<OUT>"])

**else**:

            # *Adding word's corresponding token*

**ints.append**(questnWrd2Int[word])

    # *Finally "ints" is containing a tokenized question sentence,*

    # *we append this to "questions\_to\_int"*

**questions\_to\_int.append**(ints)

# *We do same for the Answers*

answers\_to\_int = []

**for** answer **in** clean\_ans:

    ints = []

**for** word **in** **answer.split**():

**if** word **not** **in** ansWrd2Int:

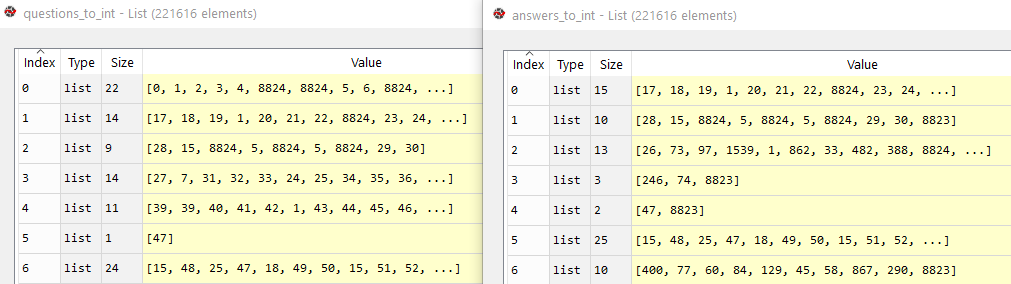
**ints.append**(ansWrd2Int["<OUT>"])

**else**:

**ints.append**(ansWrd2Int[word])

**answers\_to\_int.append**(ints)

* After running above codes we get following list of *Encoded Tokenized Sentences*.



* Each of the *Question* of the *Clean Questions List* was translated into the *unique integers* according to our *questions words* *to ans* dictionary. Same for the Answers. Those are the tokenized sentences.

Next we'll sort the tokenized sentences of questions and the answers by their length.

**4.12 Sort the Questions/Answers**

We're going to tackle the final step of our NLP phase which is to sort the both *questions* and *answers* by the *length of the questions*.

* Why:? This will *speed up the training* and help to *reduce the loss*.

Because it will reduce the amount of padding during the training.

* What is padding in NLP?

As we know all the *neural networks* needs to have the *inputs* that should be in *similar shape* and *size*. When we pre-process the texts and use the texts as an inputs for our Model. Note that *not all the sequences have the same length*, as we can say naturally some of the sequences are long in lengths and some are short. Where we know that we need to have the inputs with the same size, now here padding comes into picture. The inputs should be in same size at that time padding is necessary.

* Padding is a special form of masking where the masked steps are at the start or the end of a sequence. Padding comes from the need to encode sequence data into contiguous batches: in order to make all sequences in a batch fit a given standard length, it is necessary to pad or truncate some sequences.
* So we will have to make some *new* *lists* because we *can not change* the *order* in an *existing list*. This is not possible in python. We could override the previous list.
* But it's dangerous to do that so we'll make a new list and we'll make actually two new lists. The first one we're going to call it ***sorted\_clean\_questions*** and second one is ***sorted\_clean\_answers***.

# *Sorting the both questions and answers by the length of the questions*

max\_line\_length = 25

sorted\_clean\_questions = []

sorted\_clean\_answers = []

# *print(enumerate(questions\_to\_int));*

enumarated = **list**(**enumerate**(questions\_to\_int))

**for** length **in** **range**(1, max\_line\_length+1):

**for** i **in** **enumerate**(questions\_to\_int):

**if** **len**(i[1]) **==** length:

**sorted\_clean\_questions.append**(questions\_to\_int[i[0]])

**sorted\_clean\_answers.append**(answers\_to\_int[i[0]])

* Enumerate List of Lists and Sorting:
* **for** length **in** **range**(1, max\_line\_length+1): iterates from ***1*** to ***max\_line\_length***. This means length will first be equal to ***1***, then when the loop finishes length will be equal to ***2***, etc. until length is finally equal to ***max\_line\_length*** (25 in this case).
  + We set min length is 1, for one word questions, like what? Why? So? Etc.
  + We set max line length 25, for efficient training purpose, because too lengthy sentences are hard for training our NN-seq2seq model.
  + If this is too long, we can always reduce it. For better training we can also tune this parameter.
* **for** i **in** **enumerate**(questions\_to\_int): makes i equal to a ***tuple*** of ***(index, element)***, meaning that ***len(i[1])*** will be the length of the ***ith question*** (which is also a list in this case). As ***length*** increases in value from ***one***, the *shortest questions and answers* will gradually be added to the lists, hence the ***append()*** calls within the 2nd ***for*** loop
* The line **sorted\_clean\_questions.append**(questions\_to\_int[i[0]]) will append the question at the current index where ***len(i[1])==length***. Remember that ***i*** is a ***tuple*** of 2 elements, and the *first* is the *index* of the *current question*. So ***questions\_to\_int[i[0]]*** is the question that has length ***length***.
  + So if the *length of the question* is equal to loop-variable ***length*** then we append this question to the **sorted\_clean\_questions** list. Then we append the corresponding answers to **sorted\_clean\_answers**, using the same index.

|  |  |
| --- | --- |
| * Similarly, ***answers\_to\_int[i[0]]*** is the answer corresponding to that question. So whenever we find a slightly longer question using the ***if*** condition, the list ***sorted\_clean\_answers*** gets longer by 1 with the new answer, and ***sorted\_clean\_questions*** does as well   + Notice that we also sorted the answers by the length of the questions. That's because we want to *align* the *questions* and the *answers* to make sure that at a specific index the *answer of that index* is the *answer to the question* of that *same index*. * **enumerate()** * ***enumerate()*** in python returns a ***list*** of ***tuple***. ***enumerate()*** is applied on a list/string, to get a list of tuples where each tuple's 1st element represents the *index of enumerated lists elements* and tuple's 2nd element represents the *enumerated lists each elements*. * Since here we're dealing with *list of lists*. We'll get the following result: where tuple 1st element represents index and 2nd element is a list. |  |

* Sorted questions and corresponding answers

|  |  |
| --- | --- |
|  |  |

Now it is the end of this huge data processing phase. This is compulsory in deep NLP. Keep in mind that the work of a data scientist or machine learning scientist is 70% on data pre-processing.

Next we're going to get into *TensorFlow* to create the *encoding layer* and then the *decoding layer* in the seq2seq model.