IN4080: obligatory assignment 3

Mandatory assignment 3 consists of three parts. In Part 1, you will develop a chatbot model based on a basic retrieval-based model (25 points). In Part 2, you will implement a simple silence detector using speech processing (35 points). Finally, Part 3 will help you understand some core concepts in NLU and dialogue management through the construction of a simulated talking elevator (40 points).

You should answer all three parts. You are required to get at least 60 points to pass. The most important is that you try to answer each question (possibly with some mistakes), to help you gain a better and more concrete understanding of the topics covered during the lectures.

- We assume that you have read and are familiar with IFI's requirements and guidelines for mandatory assignments, see here and here.
- This is an individual assignment. You should not deliver joint submissions.
- You may redeliver in Devilry before the deadline (**Friday, November 11 at 23:59**), but include all files in the last delivery.
- Only the last delivery will be read! If you deliver more than one file, put them into a zip-archive. You don't have to include in your delivery the files already provided for this assignment.
- Name your submission *your_username_in4080_mandatory_3*
- You can work on this assignment either on the IFI machines or on your own computer.

The preferred format for the assignment is a completed version of this Jupyter notebook, containing both your code and explanations about the steps you followed. We want to stress that simply submitting code is **not** by itself sufficient to complete the assignment - we expect the notebook to also contain explanations (in Norwegian or English) of what you have implemented, along with motivations for the choices you made along the way. Preferably use whole sentences, and mathematical formulas if necessary. Explaining in your own words (using concepts we have covered through in the lectures) what you have implemented and reflecting on your solution is an important part of the learning process - take it seriously!

Technical tip: Some of the tasks in this assignment will require you to extend methods in classes that are already partly implemented. To implement those methods directly in a Jupyter notebook, you can use the function `setattr` to attach a method to a given class:

```
class A:
    pass
a = A()

def foo(self):
    print('hello world!')
```

```
setattr(A, 'foo', foo)
```

Part 1: Retrieval-based chatbot

We will build a retrieval-based chatbot based on a dialogue corpus of movie and TV subtitles. More specifically, we will rely on a pre-trained neural language model (BERT) to convert each utterance into a fixed-size vector. This conversion will allow us to easily determine the similarity between two utterances using cosine similarity. Based on this similarity metric, one can easily determine the utterance in the corpus that is most similar to a given user input. Once this most-similar utterance is found, the chatbot will select as response the following utterance in the corpus.

To work on this chatbot, you need to install the `sentence-transformers` package (see sentence-BERT for details):

```
$ pip install --user sentence-transformers
```

Data

We'll work with a dataset of movie and TV subtitles for English from the OpenSubtitles 2018 dataset and restricted for this assignment to comedies.

The dataset is in the gzip-compressed file `data/en-comedy.txt.gz`. The data contains one line per utterance. Dialogues from different movies or TV series are separated by lines starting with ###.

The first task will be to read through the dataset in order to extract pairs of (input, response) utterances. We want to limit our extraction to `high-quality'' pairs, and discard pairs that contain text that is not part of a dialogue, such as subtitles indicating _"(music in background)"_. We want to enforce the following criteria: 1. The two utterances should be consecutive, and part of the same movie/TV series. 2.###delimiters between movies or TV series should be discarded. 3. Pairs in which one utterance contains commas, parentheses, brackets, colons, semi-colons or double quotes should be discarded. 4. Pairs in which one utterance is entirely in uppercase should be discarded. 5. Pairs in which one utterance contains more than 10 words should be discarded. 6. Pairs in which one utterance contains a first name should be discarded (see the JSON filefirst_names.json` to detect those), as those are typically names of characters occurring in the movie or TV series.

1. Pairs in which the input utterance only contains one word should be discarded (as user inputs typically contain at least two words).

You can of course add your own criteria to further enhance the quality of the (input, response) pairs. If you want the chatbot to focus on a specific subset of movies (like TV episodes from the eighties) you are also free to do so.

Code

Here is the template code for our basic chatbot:

```
import os, random, gzip, json, re
import numpy as np
import sentence_transformers
from typing import List, Tuple, Dict

DIALOGUE_FILE= os.path.join(os.path.abspath(''), "data", "en-
comedy.txt")
FIRST_NAMES = os.path.join(os.path.abspath(''), "data",
"first_names.json")
SBERT_MODEL = "all-MiniLM-L6-v2"

class Chatbot:
    """A basic, retrieval-based chatbot"""

    def __init__(self, dialogue_data_file=DIALOGUE_FILE,
embedding_model_name=SBERT_MODEL):
        """Initialises the retrieval-based chatbot with a corpus and
an embedding model
        from sentence-transformers."""

        # Load the embedding model
        self.model =
sentence_transformers.SentenceTransformer(embedding_model_name)

        # Extracts the (input, response) pairs
        self.pairs = self._extract_pairs(dialogue_data_file)

        # Precompute the embeddings for each input utterance in the
pairs
        self.input_embeddings = self._precompute_embeddings()


    def _extract_pairs(self, dialogue_data_file:str,
max_nb_pairs:int=100000) -> List[Tuple[str,str]]:
        """Given a file containing dialogue data, extracts a list of
relevant
        (input,response) pairs, where both the input and response are
        strings. The 'input' is here simply the first utterance (such
as a question),
        and the 'response' the following utterance (such as an
answer).

        The (input, response) pairs should satisfy the following
critera:
        - The two strings should be consecutive, and part of the same
movie/TV series
```

```
        - Pairs in which one string contains commas, parentheses,
brackets, colons,
          semi-colons  or double quotes should be discarded.
        - Pairs in which one string is entirely in uppercase should be
discarded
        - Pairs in which one string contains more than 10 words should
be discarded
        - Pairs in which one string contains a first name should be
discarded
          (see the json file FIRST_NAMES to detect those).
        - Pairs in which the input string only contains one token
should be discarded.

        You are of course free to add additional critera to increase
the quality of your
        (input, response) pairs. You should stop the extract once you
have reached
        max_nb_pairs.

        """
        raise NotImplementedError()


    def _precompute_embeddings(self) -> np.ndarray :
        """Based on the sentence-BERT model in self.model, computes
the vectors associated with
        each input string of the (input, response) pairs in
self.pairs. You can ignore the response
        string of each pair. The method should return a numpy matrix
of dimension (N, d) where N is
        the number of elements in self.pairs, and d is the dimension
of the output vector
        (for instance 384).

        The vectors can be computed through the method
self.model.encode(...), which can take
        either a single string or a list of strings (note, however,
that you may receive an
        out-of-memory error if the provided list of strings is too
large for the model).
        """

        raise NotImplementedError()


    def get_response(self, user_utterance: str) -> str:
        """
        Extracts the vector for the user utterance using the sentence-
BERT model in self.model,
        and then computes the cosine similarity of this vector against
```

```
the vectors of all inputs
        in the (input, response) pairs, which should be precomputed in
self.input_embeddings.

        After computing those cosine similarity scores, the method
should find the input that
        is most similar to the user utterance, and then select the
corresponding response -- that is,
        the response in the (input, response) pair.

        You should try to implement this method using Numpy functions,
without any explicit loop.

        The method returns a string with the response of the chatbot.
        """
        raise NotImplementedError()
```

Initialisation

The initialisation of the chatbot operates in two steps. The first step, represented by the method `_extract_pairs` is to read the corpus in order to extract a list of (input, response) pairs that satisfy the seven quality criteria mentioned above. Given those pairs, the chatbot then calculates in `_precompute_embeddings` the vectors of all input utterances in those pairs using the `encode` method of the pre-trained sentence-BERT model. Note that we only need to compute the vectors for the inputs, not for the responses!

Task 1.1: Implement the method `_extract_pairs`.

```
MAX_LENGTH = 10
MIN_LENGTH = 1
```

```
def _extract_pairs(self, dialogue_data_file=DIALOGUE_FILE,
max_nb_pairs:int=100) -> List[Tuple[str,str]]:
    pass # add your implementation here
    f = open(FIRST_NAMES)

    # returns JSON object as
    # a dictionary
    data = json.load(f)
    listOfMovies=[]

    lines = open(dialogue_data_file, encoding='utf-
8').read().strip().split('\n')


    pairs = []
```

```
    i=0
    while len(pairs) < max_nb_pairs:
        inputt = lines[i].split()
        response = lines[i+1].split()
        if len(lines[i].split(' ')) < MAX_LENGTH and
len(lines[i+1].split(' ')) < MAX_LENGTH:
            if len(lines[i].split(' ')) > MIN_LENGTH and
len(lines[i+1].split(' ')) > MIN_LENGTH:
                if lines[i].isupper() == False and
lines[i+1].isupper() == False:
                    #define boolean variables that will be used to
check the presence of a name in the database
                    p0 = False
                    p1 = False
                    #check for the input
                    for l in inputt:
                        if l in data:
                            p0 = True
                    #check for the response
                    for l in response:
                        if l in data:
                            p1 = True
                    #if both False : there are no names in the pair so
we can continue sorting
                    if p0 == False and p1 == False:
                        #delete lines with punctuations
                        sorting1 = re.sub(r'[,():;""]', '', lines[i])
                        sorting2 = re.sub(r'[,():;""]', '',
lines[i+1])
                        #delete lines with punctuations
                        sorting11 = re.sub(r'\[.*?\]', '', sorting1)
                        sorting22 = re.sub(r'\[.*?\]', '', sorting2)
                        #Check if the input and the response are the
same after the sorting
                        #if it's true : create a pair
                        if lines[i]==sorting11 and lines[i+1] ==
sorting22:
                            pairs.append([lines[i],lines[i+1]])
        i+=1

    return pairs


setattr(Chatbot, '_extract_pairs', _extract_pairs)
```

Here we sort the film dialogues according to the criteria. We store in a variable the input and the response, we use different methods to see if one of the 2 lines has a name from the database. We delete the punctuation twice because I couldn't delete the [] and the () in the

same line. If the variables after filtering are the same as the input/response pair then we store it in the pair array.

Task 1.2: Implement the method `_precompute_embeddings`.

```
from sentence_transformers import SentenceTransformer, util

def _precompute_embeddings(self):
    pass # add your implementation here
    sentence_embeddings = []

    for pair in self.pairs:
        sentence_embeddings.append(pair[0])
    return self.model.encode(sentence_embeddings)
setattr(Chatbot, '_precompute_embeddings', _precompute_embeddings)

chatbot = Chatbot()
```

Runtime

Once those pairs and corresponding vectors are extracted, we only need to implement the logic for selecting the most appropriate response for a new user input. In this simple, retrieval-based chatbot, we will adopt the following strategy:

- we first extract the vector for the new user utterance using the sentence-BERT model
- we then compute the cosine similarities between this vector and the (precomputed) vectors of all inputs in the corpus
- we search for the input with the highest cosine similarity (= the utterance in the corpus that is most similar to the new user input)
- finally, we retrieve the response associated with this input utterance, and return it

Task 1.3.: Implement the method `get_response`.

```
def get_response(self, user_utterance:str):
    pass # add your implementation here

    user_sentence = self.model.encode(user_utterance)
    cosine_scores = util.cos_sim(user_sentence, self.input_embeddings)
    pairs = []
    for i in range(len(cosine_scores[0])-1):
        pairs.append({'index': i, 'score': cosine_scores[0][i]})
    #sort the scores in decreasing order
    pairs = sorted(pairs, key=lambda x: x['score'], reverse=True)
    #Takes the first value in the list that represents the input that
    #is most similar to the user utterance
    i = pairs[0]['index']
    #print("User utterance : {} \nBest response :  {} \nScore:
{:.4f}".format(user_utterance,self.pairs[i][1], pairs[0]['score']))
```

```
    return self.pairs[i][1]
setattr(Chatbot, 'get_response', get_response)
```

We apply the model to the user's input and calculate its cosine score with input_embeddings. We store the values in an array which we sort from the best score to the lowest. We can then find the highest score corresponding to the first value in the list

Technical tip: When working with numpy arrays/matrices, you should always try to avoid going through explicit loops, as looping over values of a numpy array is highly inefficient. For instance, instead of computing the dot product of each input one after the other (within a loop), you can compute the dot product of all inputs in one single dot product operation -- which is way more efficient! See here for more details on the best way to work with numpy arrays.

You can now test your setup, by loading the chatbot (this may take some time):

```
chatbot = Chatbot()
```

And getting some answers:

```
for example in ["What is your name?", "I think I am lost.", "Did you
kill him?", "Have you been to Norway before?"]:
    print("Input:", example)
    print("Response:", chatbot.get_response(example))
    print("-----")
```

```
Input: What is your name?
Response: You know what animal scare me?
-----
Input: I think I am lost.
Response: What kind of animals scare you?
-----
Input: Did you kill him?
Response: I don't know if it animal or...
-----
Input: Have you been to Norway before?
Response: - We've been together a long time.
-----
```

I did not compile with max pairs = 100000, my computer crashed

Part 2: speech processing

In this part, we will learn how we can perform basic speech processing using operations on numpy arrays! We'll work with `wav` files.

A `wav` file is technically quite simple and encodes a sequence of integers, where each integer express the magnitude of the signal at a given time slice. The number of time slices per second is defined in the frame rate, which is often 8kHz, 16kHz or 44.1kHz. So a file with 4 seconds of audio using a frame rate of 16 kHz will have a sequence of 64 000 integers. The number of bits used to encode these integers may be 8-bit, 16-bit or 32-bit

(more bits means that the quantization of the signal at each time slice will be more precise, as there will be more levels). Finally, a `wav` file may be either mono (one single audio channel) or stereo (two distinct audio channels, each with its sequence of integers).

To read the audio data from a `wav` file, you can use `scipy`:

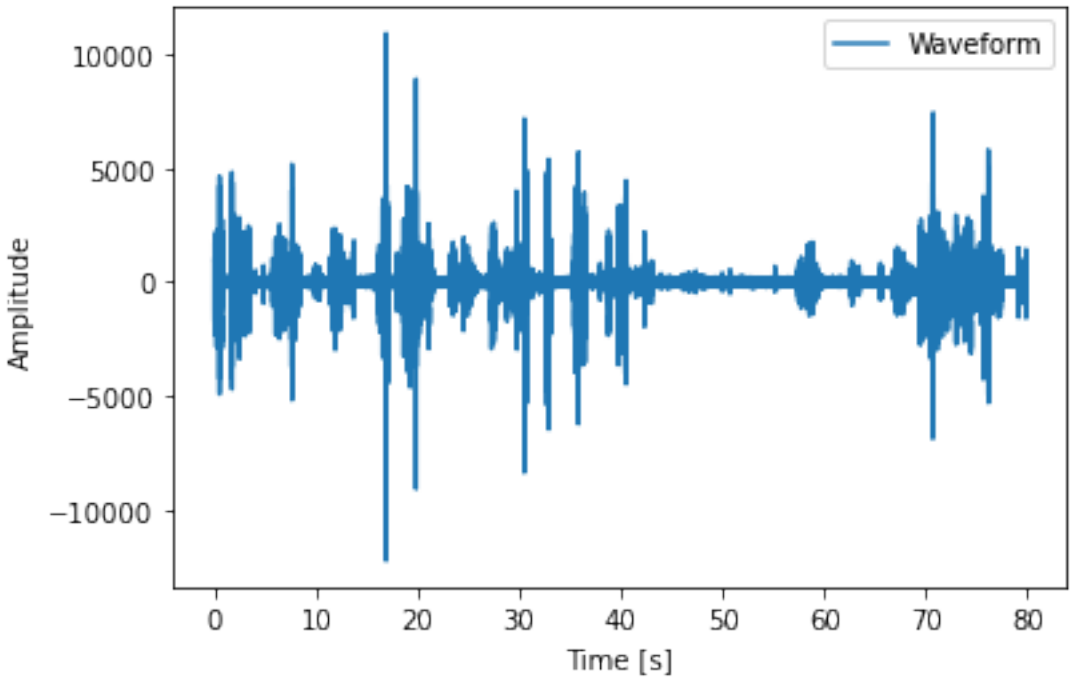
```
import scipy.io.wavfile
fs, data = scipy.io.wavfile.read("data/excerpt.wav")
data = data.astype(np.int64)
```

where `data` is the numpy array containing the actual audio data, and `fs` is the frame rate. Note that I am here converting the integers into 32-bit to avoid running into overflow problems later on (i.e.~when squaring the values).

Task 2.1: Plot (using `matplotlib.pyplot`) the waveform for the full audio clip.

```
import matplotlib.pyplot as plt
import numpy as np

#length represents the length of the audio
length = data.shape[0] / fs
time = np.linspace(0., length, data.shape[0])
plt.plot(time, data, label="Waveform")
plt.legend()
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.show()
```



For speech analysis, looking directly at the signal magnitude is typically not very useful, as we can't typically distinguish speech sounds based on the waveform. A representation of

the signal that is much more amenable to speech analysis is the *spectrogram*, which represents the spectrum of *frequencies* of a signal as it varies with time.

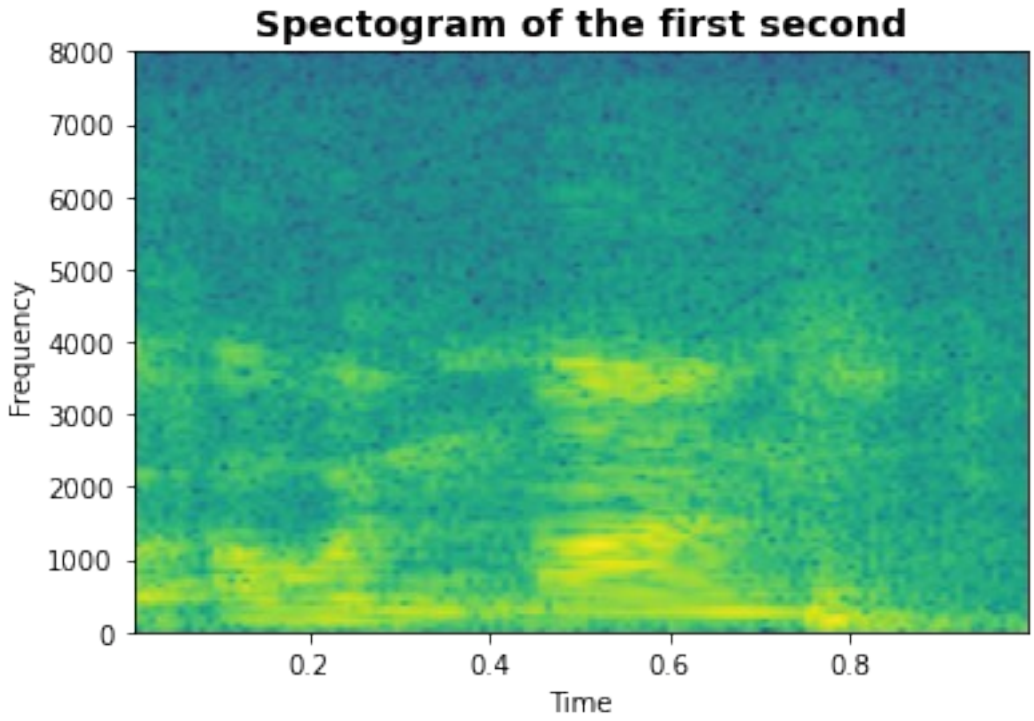
Task 2.2: Plot the spectrogram of the first second of the audio clip, using the function `matplotlib.pyplot.specgram`.

```
from scipy import signal

#We get the first second of the audio in the "first" variable that we
plot on a spectrogram
first = data[:fs]
plt.specgram(first, Fs = fs)
plt.xlabel("Time")
plt.ylabel("Frequency")
plt.title('Spectogram of the first second',
          fontsize = 14, fontweight ='bold')

plt.show()
```

```
plt.show()
```



Now, let's say we wish to remove periods of silence from the audio clip. There are many methods to detect silence (including deep neural networks), but we will rely here on a simple calculation based on the energy of the signal, which is the sum of the square of the

magnitudes for each value within a frame: \begin{equation} E = \sum_{i=1}^N X[t]^2 \ end{equation}

Task 2.3: Loop on your audio data by moving a frame of 200 ms. with a step of 100 ms., as shown in the image below. For each frame, compute the energy as in Eq. (1) and store the result.

```
import numpy

hop_length = int(fs * 0.1)

energy = []

for i in range(int(len(data)/hop_length)):
    frame = data[i*hop_length:(i+2)*hop_length]
    energy.append(sum([x**2 for x in frame]))
```

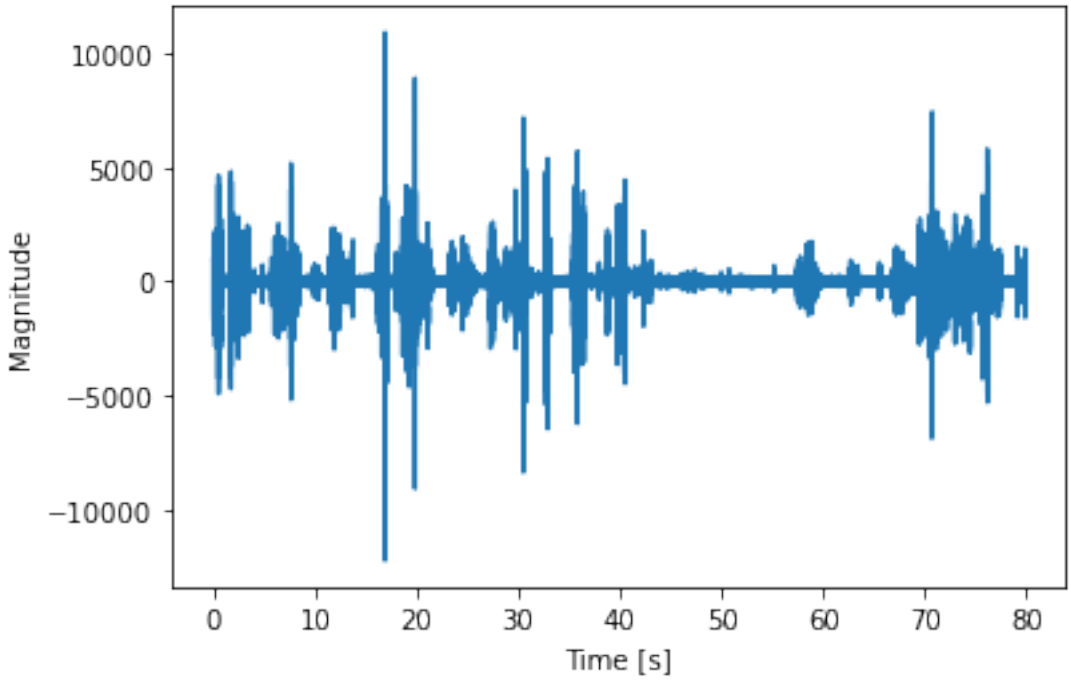
1121478915

Task 2.4: Now that we have the energy for each frame of the audio clip, we can calculate the mean energy per frame, and define silent frames as frames that have less than $\frac{1}{10}$ of this mean energy. Determine which frame is defined as silent according to this definition. Show on the plot of the waveform which segment is determined as silent, using the method `matplotlib.pyplot.hlines`.

```
from statistics import mean

#calculate the mean energy divided by 10 to set the threshold for
silent frames
mean_energy = mean(energy)/10

silent_frames = [i for i, _ in filter(lambda x: x[1] < mean_energy,
enumerate(energy))]
silent_frames = np.array(silent_frames)
plt.plot(time, data)
plt.xlabel("Time [s]")
plt.ylabel("Magnitude")
plt.hlines(np.zeros(len(silent_frames)),
(silent_frames*hop_length)/fs, (silent_frames+1)*hop_length / fs,
colors=["yellow"])
plt.show()
```



I checked several documentations, several examples and several websites but I don't understand why my graph doesn't show the silent frames

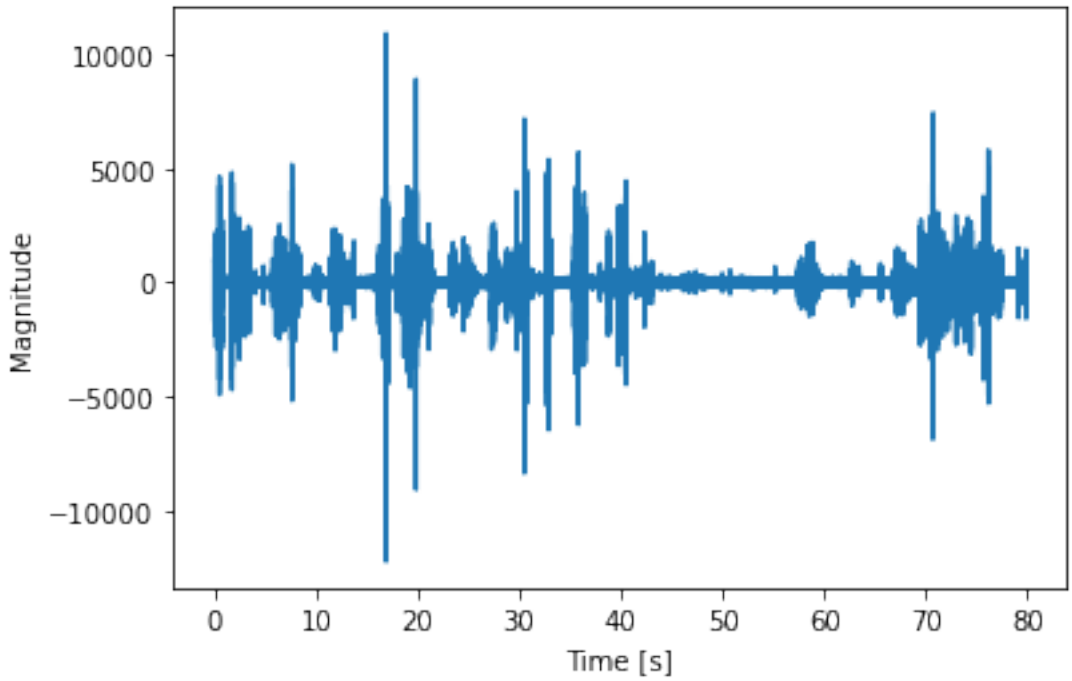
Task 2.5: If your calculations are correct, you will notice that the method has marked many short, isolated segments (e.g.~in the middle of a word) as silent. We want to avoid this of course, so we will rely on a stricter requirement to define whether a frame should be considered silent or not: we only mark a frame as silent if the frame *and all of its neighbouring frames* have less than $\frac{1}{10}$ of the mean energy. We can defines the neighbourhood of a frame as the five frames before and the five frames after. Show again on the plot of the waveform which segment is determined as silent using the stricter definition above.

```
#function that gets the neighbours of a frame and checks if they are
all silent or not

def is_silent(i, energy):
    neighbours = range(max(i-6, 0), min(i+6, len(energy)))
    compteur=True
    for neigh in neighbours:
        if neigh> mean_energy:
            compteur==False
    if compteur==True:
        return energy

silent_frames = [i for i in filter(lambda i: is_silent(i, energy),
range(len(energy)))]
silent_frames = np.array(silent_frames)
```

```
plt.plot(time, data)
plt.xlabel("Time [s]")
plt.ylabel("Magnitude")
plt.hlines(np.zeros(len(silent_frames)), silent_frames*n_div10 / fs,
(silent_frames+1)*n_div10 / fs, colors=["red"])
plt.show()
```



Task 2.5b (Optional): Modify the audio data by cutting out frames marked as silent, and using the method `scipy.io.wavfile.write` to write back the data into a `wav` file. Don't forget to convert back the data array to 16-bit using `astype(np.int16)` before saving the data to the `wav` file.

Task 2.6 (Optional, for the bravest students): Until now, we have only looked at the detection of silent frames. But what we are truly interested in is to distinguish between speech sound and the rest, which may be either silence or non-speech sounds (noise). For this, we need to analyse the audio in the *frequency domain* and look at which range of frequencies is most active within a given frame.

How do we achieve this? The key is to apply a *Fast Fourier Transform* (FFT) to go from an audio signal expressed in the time domain to its corresponding representation in the frequency domain. We can then look at frequencies in the speech range (typically between 300 Hz and 3000 Hz) and compute the total energy associated to that range. You can look here to know more about how to use `numpy` and `scipy` to perform such operations.

This task is not part of the obligatory assignment, but feel free to experiment with it if you are interested in audio processing. Students that manage to come with a solution for detecting silence and non-speech frames using FFT will get a bonus of +15 points.

Part 3: Talking Elevator

Let's assume you wish to integrate a (spoken) conversational interface to one of the elevators in the IFI building. The elevator should include the following functions:

- If the user express a wish to go to floor X (where X is an integer value between 1 and 10), the elevator should go to that floor. The interface should allow for several ways to express a given intent, such as "*Please go to the X-th floor*" or "*Floor X, please*".
- The user requests can also be relative, for instance "*Go one floor up*".
- The elevator should provide *grounding* feedback to the user. For instance, it should respond "*Ok, going to the X-th floor*" after a user request to move to X.

- The elevator should handle misunderstandings and uncertainties, e.g. by requesting the user to repeat, or asking the user to confirm if the intent is uncertain (say, when its confidence score is lower than 0.5).
- The elevator should also allow the user to ask where the office of a given employee is located. For instance, the user could ask "*where is Erik Velldal's office?*", and the elevator would provide a response such as "*The office of Erik Velldal is on the 4th floor. Do you wish to go there?*". Of course, you don't need to write down the office locations of all employees in the IFI department, we'll simply limit ourselves to the 10 names provided in the `OFFICES` dictionary (see below).
- The elevator should also be able to inform the user about the current floor (such as replying to "*Which floor are we on?*" or "*Are we on the 5th floor?*").
- Finally, if the user asks the elevator to stop (or if the user says "*no*" after a grounding feedback "*Ok, going to floor X.*"), the elevator should stop, and ask for clarification regarding the actual user intent.

Code

Here is the basic template for the implementation:

```
import re
import numpy as np
from typing import List, Tuple, Dict, Set
from elevator_utils import TalkingElevatorGUI, DialogueTurn
from sklearn.linear_model import LogisticRegression

OFFICES = {"Erik Velldal": 4, "Lilja Øvrelid":4, "Jan Tore Lønning":4,
"Nils Gruschka":9,
           "Roger Antonsen":9, "Tone Bratteteig":7, "Kristin
Bråthen":4, "Miria Grisot":6,
```

```
            "Philipp Häfliger":5, "Audun Jøsang":9}

class TalkingElevator:
    """Representation of a talking elevator. The elevator is simulated
using a GUI
    implemented in elevator_utils (which should not be modified) """

    def __init__(self):
        """Initialised a new elevator, placed on the first floor"""

        # Current floor of the elevator
        self.cur_floor: int = 1

        # (Possibly empty) list of next floor stops to reach
        self.next_stops : List[int] = []

        # Dialogue history, as a list of turns (from user and system)
        self.dialogue_history : List[DialogueTurn] = []

        # Initialises and start the Tkinter GUI
        self.gui = TalkingElevatorGUI(self)


    def start(self):
        "Starts the elevator"

        self._say_to_user("Greetings, human! What can I do for you?")
        self.gui.start()


    def process_user_input(self, user_turn : DialogueTurn):
        """Process a new dialogue turn from the user"""

        # We log the user input into the dialogue history and show it
in the GUI
        self.dialogue_history.append(user_turn)
        self.gui.display_turn(user_turn)

        # We extract the intent distribution given the user input
        intent_distrib = self._get_intent_distrib(user_turn.utterance)

        # We take into account the confidence score of that input
        intent_distrib = {intent:prob * user_turn.confidence for
intent, prob in intent_distrib.items()}

        # We extract the (possibly empty) set of detected slot values
in the input
        slot_values = self._fill_slots(user_turn.utterance)

        # We add the intent distribution and slot values to the user
```

```
turn (in case we need them)
        user_turn.intent_distrib = intent_distrib
        user_turn.slot_values = slot_values

        # Finally, we execute the most appropriate response(s) given
the
        # intent distribution and detected entities
        self._respond(intent_distrib, slot_values)


    def train_intent_classifier(self, training_data: List[Tuple[str,
str]]):
        """Given a set of (synthetic) user utterances labelled with
their corresponding
        intent, train an intent classifier. We suggest you adopt a
simple approach
        and use a logistic regression model with bag-of-words
features. This implies
        you will need to:
        - construct a vocabulary list from your training data (and
store it in the object)
        - create a small function that extracts bag-of-words features
from an utterance
        - create and fit a logistic regression model with the training
data (and store it)
        -"""

        raise NotImplementedError


    def _get_intent_distrib(self, user_input:str) -> Dict[str, float]:
        """Given a user input, run the trained intent classifier to
determine the
        probability distribution over possible intents"""

        raise NotImplementedError

    def _fill_slots(self, user_input:str) -> Dict[str,str]:
        """Given a user input, run the rule-based slot-filling
function to detect
        the occurrence of a slot value. The method returns a (possibly
empty)
        mapping from slot_name to the detected slot value, such as
{floor_number:6}.
        It is up to you to decide on the slots you want to cover in
your
        implementation"""

        raise NotImplementedError
```

```
    def _respond(self, intent_distrib: Dict[str, float], slots:
Dict[str,str]) :
        """Given an intent distribution and set of detected slots, and
the
        general state of the task and interaction (provided by the
dialogue history,
        current floor, next stops, and possibly other state variables
you might want to
        include), determine what to say and/or do, and execute those
actions. In practice,
        this method should consist of calls to the following methods:
        - _say_to_user(system_response): say something back to the
user
        - _move_to_floor(floor_number): move to a given floor
        - _stop(): stop the current movement of the elevator """

        raise NotImplementedError

    def _say_to_user(self, system_response: str):
        """Say something back to the user, and add the dialogue turn
to the history"""

        # We create a new system turn
        system_turn = DialogueTurn(speaker_name="Elevator",
utterance=system_response)

        # We add the system turn to the dialogue history and show it
in the GUI
        self.dialogue_history.append(system_turn)
        self.gui.display_turn(system_turn)


    def _move_to_floor(self, floor_number : int):
        """Move to a given floor (by adding it to a stack of floors to
reach)"""

        self.next_stops.append(floor_number)
        self.gui.trigger_movement()


    def _stop(self):
        """Stops all movements of the elevator"""

        self.next_stops.clear()
        self.gui.trigger_movement()
```

You can then simply run the elevator by starting a new `TalkingElevator` instance:

```
elevator = TalkingElevator()
elevator.start()
```

Note that you need to run Jupyter *locally* in order to start the GUI window (otherwise you may get a display error). To make things simple, the actual interface is text-based, but to simulate the occurrence of speech recognition errors, the implementation introduces some artificial noise into the user utterances (e.g.~swapping some random letters from time to time). Each user utterance comes associated with a confidence score (which would come from a speech recogniser in a real system, but is here also artificially generated).

Intent recognition

Once a new user input is received, the first step is to recognise the underlying intent -- or, to be more precise, to produce a probability distribution over possible intents.

Task 3.1: You first need to define a list of user intents that cover the kinds of user inputs you would expect in your application, such as `RequestMoveToFloor` or `Confirm`. This is a design question, and there is no obvious right or wrong answer. Define below the intents you want to cover, along with an explanation and a few examples of user inputs for each.

RequestMoveToFloor

Confirm ?

AnswerFloorNumber

Repeat ?

Task 3.2: We wish to build a classifier any user input to a probability distribution over those intents, and start by creating a small, synthetic training set. Make a list of at least 50 user utterances, each labelled with an intent defined above. You can "make up" those utterances yourself, or ask someone else to come with alternative formulations if you lack inspiration.

```
labelled_utterances = [("please i want to go to the second floor,
elevator", "RequestMoveToFloor"),
                       ("please go to the first floor fast",
"RequestMoveToFloor"),
                       ("go up to the third floor",
"RequestMoveToFloor"),
                       ("fourth floor", "RequestMoveToFloor"),
                       ("fifth floor please", "RequestMoveToFloor"),
                       ("I want to go to the sixth floor",
"RequestMoveToFloor"),
                       ("want go to seventh floor",
"RequestMoveToFloor"),
                       ("I would like to go to the eighth floor",
"RequestMoveToFloor"),
                       ("please go to the ninth floor",
"RequestMoveToFloor"),
```

```
                        ("go to the tenth floor",
"RequestMoveToFloor"),
                        ("which floor are we ?", "AnswerFloorNumber"),
                        ("am I at the fifth floor ?",
"AnswerFloorNumber"),
                        ("I would like to know where is Erik Velldal's
office ?", "AnswerFloorNumber"),
                        ("where I can find Erik Velldal's office?",
"AnswerFloorNumber"),
                        ("which floor is Lilja Øvrelid's office ?",
"AnswerFloorNumber"),
                        ("am I at Jan Tore Lønning's office ?",
"AnswerFloorNumber"),
                        ("Where is Nils Gruschka's office ?",
"AnswerFloorNumber"),
                        ("Erik Velldal, please", "RequestMoveToFloor"),
                        ("please go to the floor of Lilja Øvrelid,
elevator", "RequestMoveToFloor"),
                        ("Jan Tore Lønning, elevator",
"RequestMoveToFloor"),
                        ("please go to the floor of Nils Gruschka",
"RequestMoveToFloor"),
                        ("which floor is the floor of Roger
Antonsen ?", "AnswerFloorNumber"),
                        ("I want to go to the floor of Tone Bratteteig,
elevator", "RequestMoveToFloor"),
                        ("where i can find Kristin Bråthen's floor ?",
"AnswerFloorNumber"),
                        ("please go to the floor of Miria Grisot",
"RequestMoveToFloor"),
                        ("I wanna go to Philipp Häfliger, elevator",
"RequestMoveToFloor"),
                        ("where is the floor of Audun Jøsang,
elevator", "AnswerFloorNumber"),
                        ("please go to the floor 1, elevator",
"RequestMoveToFloor"),
                        ("go to the floor 2", "RequestMoveToFloor"),
                        ("go up to the floor 3, elevator",
"RequestMoveToFloor"),
                        ("please go to the floor 4, elevator",
"RequestMoveToFloor"),
                        ("where is floor number 5 ?",
"AnswerFloorNumber"),
                        ("please go to the floor 6, elevator",
"RequestMoveToFloor"),
                        ("go to floor number 7", "RequestMoveToFloor"),
                        ("please go to the floor 8",
"RequestMoveToFloor"),
                        ("floor 9, elevator", "RequestMoveToFloor"),
                        ("floor 10 please", "RequestMoveToFloor"),
```

```
                        ("go floor one", "RequestMoveToFloor"),
                        ("please go to the next floor, elevator",
"RequestMoveToFloor"),
                        ("I want to go to the floor number one,
elevator", "RequestMoveToFloor"),
                        ("please go to the floor number two, elevator",
"RequestMoveToFloor"),
                        ("am I at the floor number three, elevator ?",
"AnswerFloorNumber"),
                        ("please go to the floor number four,
elevator", "RequestMoveToFloor"),
                        ("go floor number five", "RequestMoveToFloor"),
                        ("go up to the floor number six",
"RequestMoveToFloor"),
                        ("please go to the floor number seven,
elevator", "RequestMoveToFloor"),
                        ("go up to the floor number eight, elevator",
"RequestMoveToFloor"),
                        ("floor number 9", "RequestMoveToFloor"),
                        ("I would like to go to the floor number 10",
"RequestMoveToFloor"),
                        ]
```

Task 3.3: The next step is to train the intent classifier, by implementing the method `train_intent_classifier`. To make things simple, we will stick to bag-of-word features. This means you will need to

- define a vocabulary list from the (tokenized) training data defined above. This step is necessary to define the words that will be considered in the "bag-of-words" features.
- implement a small feature extraction method that takes an utterance as input and output a feature vector (bag-of-words)
- create the logistic regression model and fit it to the training data. We suggest you rely on the `LogisticRegression` from `scikit-learn`. To improve the model predictions, we also recommend to turn off the regularization by setting `penalty='none'` when instantiating the model.

```
from sklearn.feature_extraction.text import CountVectorizer

def train_intent_classifier(self, training_data: List[Tuple[str,
str]]):
    pass # add your implementation here

    utterances = []
    intents = []
    for data in training_data:
        utterances.append(data[0])
        intents.append(data[1])

    vectorize = CountVectorizer(token_pattern=r"(?u)\b\w+\b")
```

```
    self.word_tokenizer = vectorize.build_tokenizer()

    self.features = set([word.lower() for sent in utterances for word
in self.word_tokenizer(sent)])

    vector = np.empty((len(utterances), len(self.features)))
    for utterance in utterances:
        np.append(vector, [self.createBagOfWords(utterance)], axis=0)

    self.model = LogisticRegression(penalty="none").fit(vector,
intents)


setattr(TalkingElevator, 'train_intent_classifier',
train_intent_classifier)

test = TalkingElevator()

test.train_intent_classifier(labelled_utterances)
```

```
def createBagOfWords(self, utterance) -> np.ndarray:
    vect = np.zeros(len(self.features))

    for word in self.word_tokenizer(utterance):
        if word in self.features:
            vect[list(self.features).index(word)] += 1
    return vect

setattr(TalkingElevator, 'createBagOfWords', createBagOfWords)
```

Task 3.4: Now that your intent classifier is trained, implement the method `_get_intent_distrib` that takes a new user utterance as input, and outputs a probability distribution over intents.

```
def _get_intent_distrib(self, user_input:str) -> Dict[str, float]:
    vectorize = CountVectorizer(token_pattern=r"(?u)\b\w+\b")
    v = vectorize.fit_transform([user_input])
    proba = self.model.predict_proba(v)[0]
    dictt = {
      "intend": user_input,
      "probabily": proba
    }
    return dictt

setattr(TalkingElevator, '_get_intent_distrib', _get_intent_distrib)
```

Slot filling

In addition to the intents themselves, we also wish to detect some slots, such as floor numbers or person names. For this step, we will not use a data-driven model, but rather rely on a basic, rule-based approach:

- For floor numbers, we will rely on string matching (with regular expressions or basic string search) that detect patterns such as "X floor" (where X is [first,second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth]) or "floor X" (where X is between 1 and 10).
- For person names, we will simply define a list of person names (you do not have to use the full names of IFI employees, just 4-5 names will suffice) to detect and search for their occurrence in the user input.

The results of the slot filling should be a dictionary mapping slot names to a canonical form of the slot value. For instance, if the utterance contains the expression "ninth floor", the resulting slot dictionary should be `{"floor_number":9}`.

Task 3.5: Implement the method `_fill_slots` that will detect the occurrence of those slots in the user input.

```
def _fill_slots(self, user_input:str) -> Dict[str,str]:
    pass # add your implementation here

    number = ['1','2','3','4','5','6','7','8','9','10']
    numberLetter = {'one':1,
'two':2,'three':3,'four':4,'five':5,'six':6,'seven':7,'eight':8,'nine'
:9,'ten':10}
    numberTh =
{'first':1,'second':2,'third':3,'fourth':4,'fifth':5,'sixth':6,'sevent
h':7,'eighth':8,'nineth':9,'tenth':10}
    names={"Velldal":4,'Øvrelid':4,'Lønning':4}
    inputUser = user_input.split()
    for y in inputUser:
        y = re.sub(r'[^\w\s]', '', y)
        if y in number:
            return {"floor_number" : y}
        elif y in numberLetter:
            return {"floor_number" : numberLetter[y]}
        elif y in numberTh:
            return {"floor_number" : numberTh[y]}
        elif y in names:
            return {"floor_number" : names[y]}

setattr(TalkingElevator, '_fill_slots', _fill_slots)

test = TalkingElevator()

inpu = 'i want to go to the floor 9'
test._fill_slots(inpu)
```

```
{'floor_number': '9'}
```

Response selection

Finally, the last step is to implement the response selection mechanism. The response will depend on various factors:

- the inferred user intents from the user utterance
- the detected slot values in the user utterance (if any)
- the current floor
- the list of next floor stops that are yet to be reached
- the dialogue history (as a list of dialogue turns).

The response may consist of verbal responses (enacted by calls to `_say_to_user`) but also physical actions, represented by calls to either `_move_to_floor` or `_stop`.

Task 3.6: Implement the method `respond`, which is responsible for selecting and executing those responses. The responses should satisfy the aforementioned conversational criteria (provide grounding feedback, use confirmations and clarification requests etc.).

```
def _respond(self, intent_distrib: Dict[str, float], slots:
Dict[str,str]) :
    pass # add your implementation here
    self._say_to_user('test')
    self.dialogue_history.append(user_turn)
    self.gui.display_turn(user_turn)
    slots = self._fill_slots("floor nine")
setattr(TalkingElevator, '_respond', _respond)
```

We are now ready to test our prototype, which can be run like this:

```
elevator = TalkingElevator()
elevator.train_intent_classifier(labelled_utterances)

elevator.start()
```

Again, there is no obvious right/wrong answer for this exercise -- the main goal is to reflect on how to design conversational interfaces in a sensible manner, in particular when it comes to handling uncertainties and potential misunderstandings.