oblig3

November 4, 2023

1 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 6 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)
```

1.1 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
```

1.1.1 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 file`first_names.json‘ to detect those), as those are typically names of characters occurring in the movie or TV series. 7. 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.

1.1.2 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.gz")
     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()
```

1.1.3 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`.

[ ]:
```
# Punctuations i dont want to include:
punctuation = [',', '(', ')', '[', ']', ';', ':', '"', '###']

# Opening and reading the file FRST_NAMES
with open(str(FIRST_NAMES), "r") as first_names_file:
        first_names = json.load(first_names_file)

# function checking if he pairs are valid
def validity_pair(pair):
    for utterance in pair:
        Tokens = utterance.split()
        for word in Tokens:
            #check for word being firstname
            if word in first_names:
                 return False
        #check for utterence having length one
        if len(Tokens)==1:
            return False
         #Check for puctuations
        for char in punctuation:
            if (char in utterance):
                return False
        #Check for capitalised
        if utterance.isupper():
            return False
        #Check for sentence with more than 10 words
        if (len(Tokens)> 10):
            return False

    return True
```

```
[ ]: def _extract_pairs(self, dialogue_data_file:str, max_nb_pairs:int=10000) ->␣
      ↪List[Tuple[str,str]]:
         pairs = []
         previous_line = None
         current_line = None
         with gzip.open(str(dialogue_data_file), "rt") as f:
             for line in f:
                 decoded_line = line.strip()
                 current_line = decoded_line
                 if previous_line == None:
                     previous_line = decoded_line
                 else:
                 # keeping the strings to be part of the same movie
                     if validity_pair((previous_line, current_line)):
                         if len(pairs) < max_nb_pairs:
                             pairs.append((previous_line, current_line))
                             previous_line = current_line
                         else:
                             break
                     else:
                         previous_line = current_line

         return pairs

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

Task 1.2: Implement the method _precompute_embeddings.

```
[ ]: def _precompute_embeddings(self):
         pair_list = list(pair[0] for pair in self.pairs)
         return self.model.encode(pair_list)

     setattr(Chatbot, '_precompute_embeddings', _precompute_embeddings)
```

1.1.4 Search

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`.

```
[ ]: from numpy.linalg import norm

     def get_response(self, user_utterance:str):
         ref_vec = self.model.encode(user_utterance)
         similarities = np.dot(ref_vec, self.input_embeddings.T)
         most_similar_idx = np.argmax(similarities)
         return self.pairs[most_similar_idx][1]

     setattr(Chatbot, 'get_response', get_response)
```

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: We can't use our own names right?
-----
Input: I think I am lost.
Response: Have you seen my new nurse?
-----
Input: Did you kill him?
Response: He went outside.
-----
Input: Have you been to Norway before?
Response: Why didn't you wake me up like you usually do?
-----
```

1.1.5 Better performance

Task 1.4 (optional, +10 points): Adapt the code you wrote to rely on *approximate search* with the FAISS library instead of exact search. You can have a look at the code here for inspiration. Use

%%timeit to measure the performance improvements compared to the exact search you implemented in Task 1.3.

1.2 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).

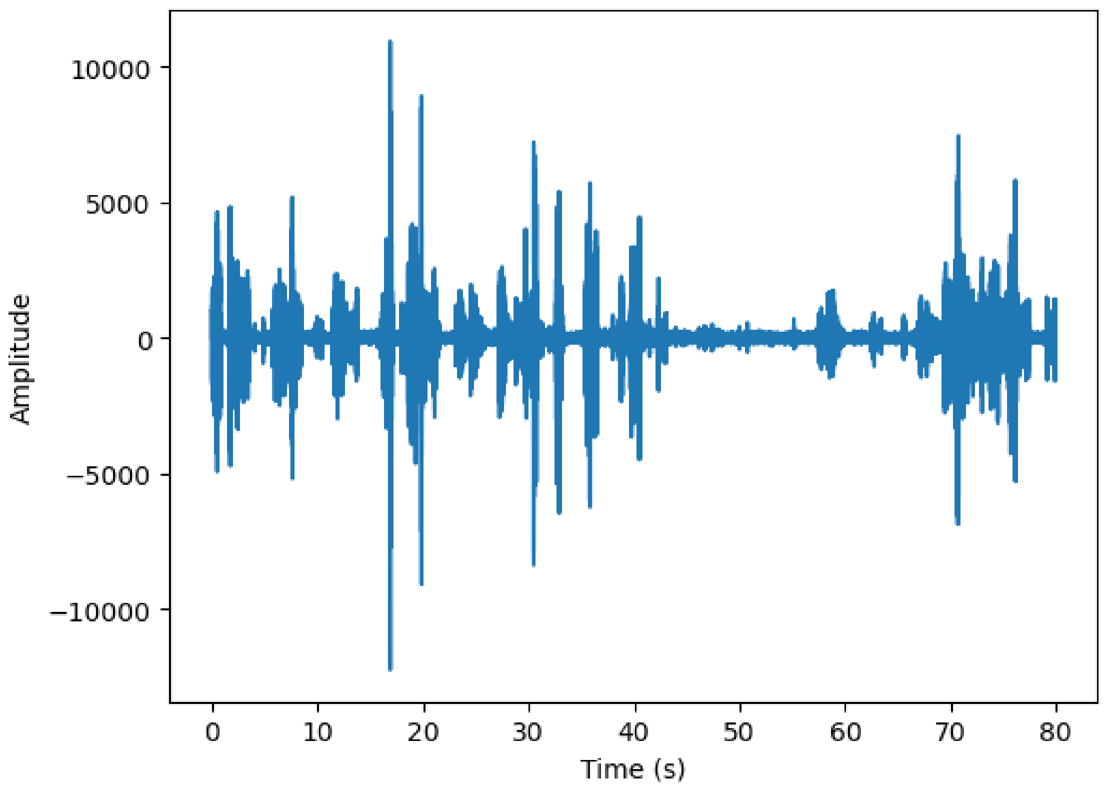
To read the audio data from a wav file, you can use scipy:

[ ]:
```
import scipy.io.wavfile
fs, data = scipy.io.wavfile.read("data/excerpt.wav")
data = data.astype(np.int32)
```

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.

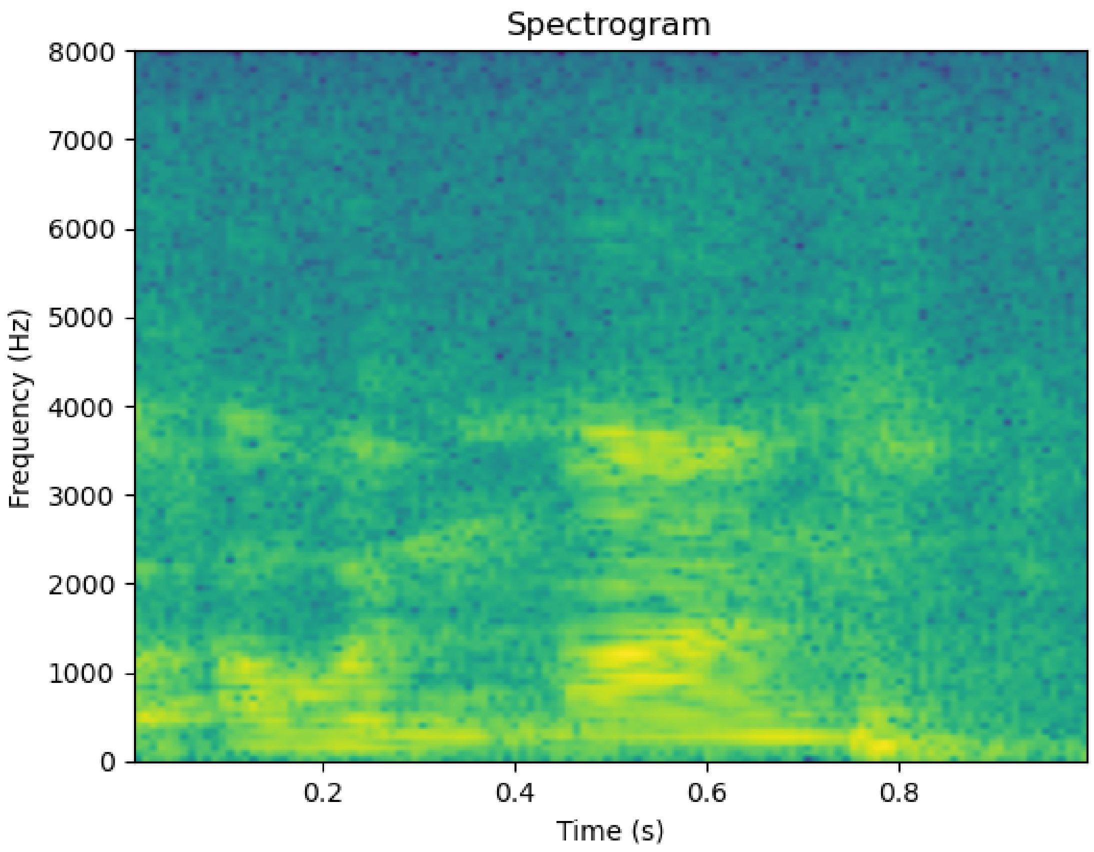
[ ]:
```
import matplotlib.pyplot as plt

length = data.shape[0] / fs
time = np.linspace(0, length, data.shape[0])
plt.plot(time, data)
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`.

[ ]:
```
plt.specgram(data[:fs], Fs=fs, NFFT=256, noverlap=128, scale='dB')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.title('Spectrogram')
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:

$$E = \sum_{i=1}^{N} X[t]^2 \tag{1}$$

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.

```
[ ]: array_len = len(data)
     offset = int(0.1*16000)
     sum_len = int(0.2*16000)

     energy = []
     for start in range(0, array_len, offset):
         energy.append(np.sum(data[start: start + sum_len]**2))

     energy = np.array(energy)
```

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`.

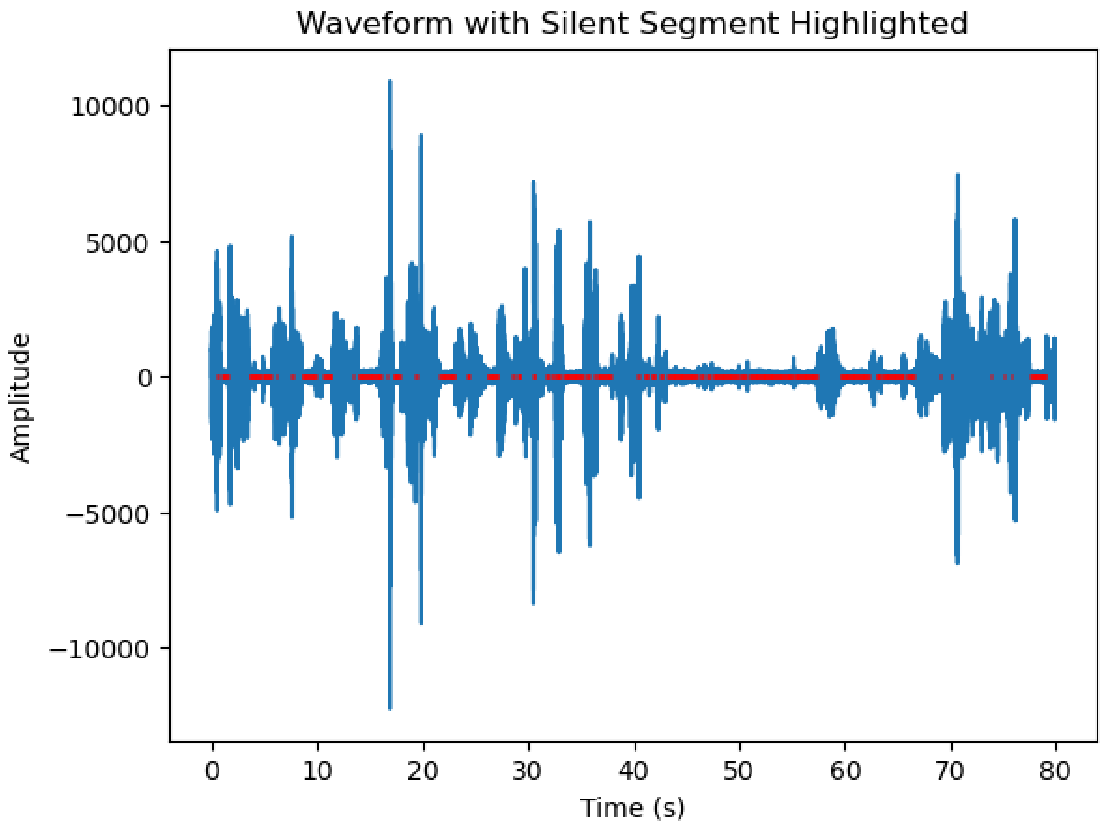
```
[ ]: energy_mean = np.mean(energy)
     energy_mean
```

```
[ ]: 232422276.2390488
```

```
[ ]: # Determining and defining silent frames
     threshold = energy_mean/10
     silent_frames = np.array([i for i, frame in enumerate(energy) if frame <␣
      ↪threshold])
     len(silent_frames)
```

```
[ ]: 381
```

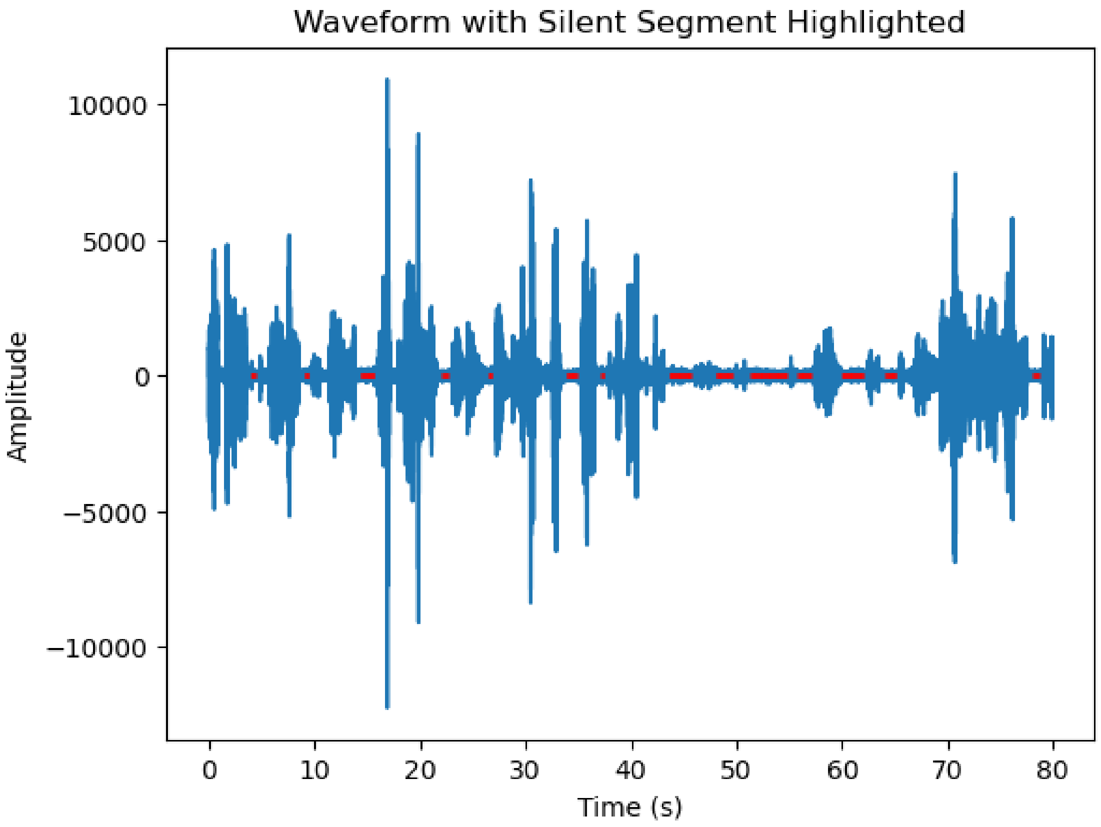
```
[ ]: # Showing which segment is determined as silent

     length = data.shape[0] / fs
     time = np.linspace(0, length, data.shape[0])
     plt.plot(time, data)

     for frame in silent_frames:
         plt.hlines(0, time[frame * offset], time[frame*offset+sum_len], colors='r',␣
      ↪linewidth=2)

     plt.title("Waveform with Silent Segment Highlighted")
     plt.xlabel("Time (s)")
     plt.ylabel("Amplitude")
     plt.show()
```



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.

```
[ ]: # Determining and defining silent frames with the consideration of neighbours

     threshold = energy_mean/10

     silent_frames_neighbours = []
     for i in range(5, len(energy)-5):
         if all(energy[i:i+5] < threshold) and all(energy[i-5: i] < threshold):
             silent_frames_neighbours.append(i)

     len(silent_frames_neighbours)
```

```
[ ]: 156
[ ]: # Showing which segment is determined as silent using neighbour method

     length = data.shape[0] / fs
     time = np.linspace(0, length, data.shape[0])
     plt.plot(time, data)

     for frame in silent_frames_neighbours:
         plt.hlines(0, time[frame * offset], time[frame*offset+sum_len], colors='r',␣
      ↪linewidth=2)

     plt.title("Waveform with Silent Segment Highlighted")
     plt.xlabel("Time (s)")
     plt.ylabel("Amplitude")
     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.

```
[ ]: # Using the silent frame date which considers the neighbours to edit the data

     import sounddevice as sd

     data_new = data.copy()

     for frame in silent_frames_neighbours:
         data_new = np.delete(data_new, np.s_[frame * offset : frame*offset+sum_len])

     data_new = np.int16(data_new)


     scipy.io.wavfile.write("data/excerpt_modify.wav", fs, data_new)
```

```
[ ]: # Using silent frame data without considering neighbours to edit the audio

     data_new_new = data.copy()

     for frame in silent_frames:
         data_new_new = np.delete(data_new_new, np.s_[frame * offset :␣
      ↪frame*offset+sum_len])

     data_new_new = np.int16(data_new_new)
```

```
[ ]: sd.play(data_new_new, fs)
     sd.wait()
```

Task 2.6 (Optional, + 15 points): 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.

1.3 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.

1.3.1 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
from sklearn.feature_extraction.text import CountVectorizer


OFFICES = {"Erik Velldal": 4, "Lilja Øvrelid":4, "Yves Scherrer":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)

        #Countverctorizer
        self.cv = CountVectorizer()
        #Classifier
        self.clf = LogisticRegression()


    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).

1.3.2 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.

1. intent: RequestMoveToFloor
    - Explanation: Users express their desire to move to a specific floor
    - Examples:
        - "Floor seven, please."
        - "Take me to the 3rd floor, please."
        - "Going to the top floor"
2. Intent: Complain
    - Explanation: User express dissatisfaction or concerns about the elevator.
    - Examples:
        - "Hurry up, I'm running late!"
        - "This elevator is so slow"
        - "I am in a hurry, can we skip some floors?"
3. Intent: Greeting
    - Explanation: User initiate a friendly conversation with the elevator.
    - Examples:
        - "Good morning"
        - "Hello, how are you?"
4. Intent: Smalltalk
    - Explanation: Users engage in casual small talk with the elevator
    - Examples:
        - "Isn't it a nice weather today?"
        - "What an aweful day, isn't it?"
5. Intent: Emergency
    - Explanation: Users seek help or indicate an emergency situation.
    - Examples:
        - "Can you call for assistance?"
        - "Send help"
6. Intent: Direction
    - Explanation: Users ask for direction or inquire about office locations.
    - Examples:
        - "Where is Erik Velldal's office"
        - "Whose offices are at fourth floor?"
7. Intent: Confirm
    - Explanation: Users confirm or affirm a previous statement or question.
    - Examples:
        - "Yes"
8. Intent: Stop
    - Explanation: USers request the elevator to stop its actions
    - Examples:
        - "Stop!"
        - "Hold on!"
9. Intent: Door
    - Explanation: Users make requests related to the elevator doors
    - Examples:
        - "Please hold the door"

    – “Hold the door”
10. Intent: Politeness
    • Explanation: Users engage in polite conversation or express gratitude
    • Examples:
        – “Good morning!”
        – “Good evening!”
11. intent: Features
    • Explanation: Users inquire about or request specific features of the elevator
    • Examples:
        – “Can you turn on the music?”
        – “Can you switch off the lights?”

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 go to the second floor, elevator",␣
      ↪"RequestMoveToFloor"),
                              ("Floor seven, please.", "RequestMoveToFloor"),
                              ("Take me to the 3rd floor, please.",␣
      ↪"RequestMoveToFloor"),
                              ("Going to the top floor", "RequestMoveToFloor"),
                             ("Go one floor up", "RequestMoveToFloor"),
                             ("Going one floor down", "RequestMoveToFloor"),
                             ("I'd like to go to the 6th floor, please",␣
      ↪"RequestMoveToFloor"),
                             ("Hurry up, I'm running late!", "Complain"),
                             ("This elevator is so slow", "Complain"),
                             ("I am in a hurry, can we skip some floors?",␣
      ↪"Complain"),
                             ("Can you slow down?", "Complain"),
                             ("Good morning", "Greeting"),
                             ("Hello, how are you?", "Greeting"),
                             ("Isn't it a nice weather today?", "SmallTalk"),
                             ("What an aweful day, isn't it?", "SmallTalk"),
                             ("Nice interior design in this elevator, isn't it?",␣
      ↪"SmallTalk"),
                             ("Can you call for assistance?", "Emergency"),
                             ("Is there an emergency button?", "Emergency"),
                             ("Send help", "Emergency"),
                             ("SOS", "Emergency"),
                             ("Where is Erik Velldal's office", "Direction"),
                             ("Take me to Lilja Øvrelid office", "Direction"),
                             ("Whose offices are at fourth floor?", "Direction"),
                             ("Going up or down?", "Direction"),
                             ("How many floors are there in the building?",␣
     ↪"Direction"),
                             ("Can you direct me to the nearest restroom?",␣
     ↪"Direction"),
                             ("How do I get to Audun Jøsang office?", "Direction"),
                             ("Which floor are we on?", "Direction"),
                             ("Are we on the 5th floor?", "Direction"),
                             ("Is this the second floor?", "Direction"),
                             ("No", "Stop"),
                             ("Yes", "Confirm"),
                             ("Stop", "Stop"),
                             ("Hold on!", "Stop"),
                             ("Wait a minute", "Stop"),
                             ("Please hold the door", "Door"),
                             ("Please close the door", "Door"),
                             ("Please open the door", "Door"),
                             ("Open the door", "Door"),
                             ("Close the door", "Door"),
                             ("Hold the door", "Door"),
                             ("Good morning!", "Politeness"),
                             ("Thank you!", "Politeness"),
                             ("Good evening!", "Politeness"),
                             ("Good night", "Politeness"),
                             ("Can you turn on the music?", "Features"),
                             ("Are there restrooms on this floor?", "Features"),
                             ("Is this elevator handicap-accessible?", "Features"),
                             ("Can you switch off the lights?", "Features"),
                             ("Is there a lost and found?", "Features")
                            ]

      print(len(labelled_utterances))
```

```
50
```

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 nltk.tokenize import word_tokenize


def train_intent_classifier(self, training_data: List[Tuple[str, str]]):
    utternaces, labels = zip(*training_data)
    VocabList = [word_tokenize(i) for i in utternaces]
    x_train = [' '.join(token) for token in VocabList]

    #feature extraction

    x_train_bow = self.cv.fit_transform(x_train)

    # Creating logreg and fitting it to the model.
    self.clf = LogisticRegression(penalty=None)
    self.clf.fit(x_train_bow, labels)



setattr(TalkingElevator, 'train_intent_classifier', train_intent_classifier)
```

Note: if you wish, instead of a bag-of-words classifier, you can try running a LLM such as the LLama 27b model optimized for chat (see here for the HuggingFace model) and provide training examples in the prompt (= *in-context learning*).

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]:

   x_test_bow = self.cv.transform([user_input])
   intent_dist = self.clf.predict_proba(x_test_bow)[0]
   return{intent:prob for intent, prob in zip(self.clf.classes_, intent_dist)}

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

1.3.3 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.

```
[ ]: import string

     def _fill_slots(self, user_input:str) -> Dict[str,str]:
         digits = {"first":1, "second":2, "third":3, "fourth":4, "fifth":5, "sixth":
      ↪6, "seventh":7, "eighth":8, "ninth":9, "tenth": 10,"one":1, "two":2, "three":
      ↪3, "four":4, "five":5, "six":6, "seven":7, "eight":8, "nine":9, "ten": 10}
         slot_dict = {}
         translator = str.maketrans('','', string.punctuation)
         user_input_person = user_input.translate(translator).split()
         user_input = user_input.lower().split()

         if "floor" in user_input:
             for char in user_input:
                 if char.isdigit():
                     slot_dict["floor_nr"] = str(char)
                     break
                 elif char in digits.keys():
                     slot_dict["floor_nr"] = str(digits[char])
                     break

         elif "up" in user_input:
             floor = self.cur_floor + 1
             slot_dict["floor_nr"] = str(floor)
         elif "down" in user_input:
             floor = self.cur_floor - 1
             slot_dict["floor_nr"] = str(floor)


         for key in OFFICES.keys():
             for i in range(len(user_input_person)-1):
                 if key in user_input_person[i]+ ' ' + user_input_person[i+1]:
                     person = user_input_person[i] + ' ' + user_input_person[i+1]
                     slot_dict["Person"] = str(person)
                     break

         return slot_dict

     setattr(TalkingElevator, '_fill_slots', _fill_slots)
```

1.3.4 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]) :
         """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 """

         likely_intent = max(intent_distrib, key=intent_distrib.get)

         if likely_intent == "RequestMoveToFloor":
             if "floor_nr" in slots:
                 if slots["floor_nr"] != self.cur_floor:
                     self._move_to_floor(int(slots["floor_nr"]))
                     self._say_to_user(f"Ok. Moving to the {slots['floor_nr']}-th␣
      ↪floor.")
                 else:
                     self._say_to_user("We are already on that floor")
             else:
                 self._say_to_user("Could you repeat the floor number you would like␣
      ↪to go to?")
         elif likely_intent == "Complain":
             self._say_to_user("To lose patience is to lose the battle!")
         elif likely_intent == "Greeting":
             self._say_to_user("I am fine thank you! How are you doing today?")
         elif likely_intent == "SmallTalk":
             self._say_to_user("Yes I can relate. Tell me about it")
         elif likely_intent == "Emergency":
             self._say_to_user("I have called the emergancy for you now. Please keep␣
      ↪calm.")
         elif likely_intent == "Direction":
             if "Person" in slots:
                 person = slots["Person"]
                 floor = OFFICES[person]
                 self._say_to_user(f"The office of {person} is at floor {floor}. Do␣
      ↪you wish to got there?")
             else:
                 self._say_to_user("Whose office are you looking for?")
```

```
    elif likely_intent == "Confirm":
        self._say_to_user("Then I will proceed.")
    elif likely_intent == "Stop":
        self._stop()
    elif likely_intent == "Door":
        self._say_to_user("Sure!")
    elif likely_intent == "Politeness":
        self._say_to_user("Hello! How are you?")
    elif likely_intent == "Features":
        self._say_to_user("Of course!")

    #Handling misunderstanding:
    elif intent_distrib[likely_intent] < 0.5:
        self._say_to_user("Could you please reformulate your request?")

    else:
        self._say_to_user("Could you please reformulate your request?")


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.