POLITECNICO OF TURIN

Master's Degree in Computer engineering - software

Master's Degree Thesis

Applying Machine Learning to a Telegram Conversational Agent as Educational Support

Supervisors

Prof. Marco TORCHIANO

Dr. Simone LEONARDI

Dr. Isabeau OLIVERI

Candidate

Federico Silvio GORRINO

October 2020

Summary

In this thesis the problem of communication between teacher and student was addressed.
At present, if the student needs it, communications between these people take place via email, is there a way to improve this interaction?
At the request of Professor Torchiano, I tackled the research and I created a telegram bot that, through linguistic analysis, can make this experience better.
In the next few pages I will go through all the steps I have gone through and all the research I have done to arrive at the final solution.

Table of Contents

Chapter 1

Introduction

1.1 The problem

One of the most frustrating and tedious parts of the university career is certainly the communication between students and professors. It happens many times via email, but this means is not at all optimal for several reasons:

- It is frustrating for the professor to receive many similar emails and would need a more efficient tool to send an answer only when a completely new question arises.
- The student sometimes has to wait an exorbitant amount of time for very simple questions.

1.2 Change of technology

To optimize this process I was asked to develop a telegram bot, that is, to use a technology very different from the usual email exchange.
The initial idea was to use the months of December, January and February for the development and then subsequently to do a testing phase during the second teaching period but the covid-19 has made work and communications much more difficult and therefore it is everything became very difficult.

1.3 The idea

The main idea suggested to me by Professor Torchiano was to save all the questions asked by the students in a database and use a linguistic correlation mechanism to understand if a question has already been asked or not.
When a question is sent the program must follow the following algorithm:

```
    def new_question(question):
        if question in database and get_response(question)!=None:
            send_notify_to_the_student()
        elif question not in database:
            send_notify_to_the_teacher()
        elif question in database and get_response(question)==None:
            pass
```

Obviously the flow, going forward with the project, became complicated and more steps and more notifications for users were added.

1.4 Covid-19

As I said earlier, covid-19 played a fundamental role in writing this thesis. I added functionality to the initial project assuming massive use of the bot in the 20/21 teaching year.
I hypothesized a year that was still very much based on online teaching.

1.5 Sections and citations

Many times I will find myself talking in the same section of the same article or I will get info to write the section from many parts of the same web page. In these cases, in order not to quote the source a thousand times, I will put the quote next to the section title.

1.6 The thesis code

The python code of the work can be found on the following page.[1]
The database instead you can find it by following this link.[2]

Chapter 2

Chat applications

Telegram was chosen for this project, but now let's see a description of the famous messaging app:

2.1 Telegram[3]

Telegram is a cloud-based instant messaging and broadcasting service. The official Telegram clients are distributed as free software for Android, GNU / Linux, iOS, MacOS, Windows NT and Windows Phone. Features of Telegram are the ability to exchange text messages between two users or between groups of up to 200,000 participants, make "point-to-point" encrypted voice calls, exchange voice messages, video messages, photographs, videos, stickers and files of any type up to 2.0 GB and optionally send point-to-point encrypted text messages between two users (not in group chat). Through the channels it is also possible to broadcast live audio / video and text to the members who join.
To understand how telegram works we need to analyze its components:

- **Users and messages**: The messages sent are saved on the Telegram cloud, so as to guarantee instant synchronization. The result allows the user to be able to access messages from several devices at the same time, including tablets and computers. Subsequently, new features were introduced that allow you to delete sent and received messages and that allow you to modify the messages sent. When registering, the user can choose whether to allow others to search for him by entering the nickname chosen during registration preceded by @ in the search bar. This optional feature allows others to search for you on Telegram without necessarily knowing your phone number.

- **Chats**:

  - **Cloud chats**: The chat uses client-server encryption, that is, it is encrypted from the device to the Telegram servers and vice versa. Therefore, the conversation should remain encrypted on the Telegram servers in order to be synchronized between multiple devices (e.g. smartphone, tablet, PC). The classic chat allows the sending of text messages, voice messages, video messages, current GPS position, GPS coordinates on the map, contacts and any type of file with a maximum size of 2.0 GB.
  - **Secret chats**: The chat uses end-to-end encryption (or called point-to-point), that is, it is encrypted between the two devices involved in the conversation. Consequently, the conversation is not saved on the Telegram servers. While these conversations are considerably safer from a privacy point of view, on the one hand you should know that consequently the chat cannot be synchronized between multiple devices but can only be viewed from the device from which it was started. Secret chats use Diffie-Hellman key exchange and allow you to set a "self-destruct timer", ie a certain time interval after which a message, after it has been viewed by the recipient, is deleted. In Secret Chats you cannot take screenshots, but if you succeed, a notification will be sent to the other user.

- **Groups**: Telegram groups can contain up to 200,000 members, you can set administrators with selectable permissions. By default in groups, new members cannot see all message history unless it is enabled by settings or the group is made public.

- **Channels**:Channels are chats where anyone who is an administrator can send messages to channel members, even if they cannot reply or comment.

- **Bots**: Telegram has introduced a platform to allow third party developers to create Bots. Bots are Telegram accounts, managed by a program, which offer multiple features with immediate and fully automated responses. The inline mode for bots, which allows you to use a bot by simply mentioning it with your username in any chat. Botfather (@BotFather) was created to create and manage their bots.

- **Stickers**: Stickers are high definition images saved on the cloud. They are intended to make emojis and messages more expressive and compelling. The Stickers are grouped into sets, called "Sticker Packs" which can be added to your library via a link or by clicking on a sticker sent by another user and then clicking "Add Sticker". Each set can contain a maximum of 120 stickers. Telegram offers a pre-installed sticker package, with the possibility for users to create others via the official @Stickers bot.

2.2 Other applications

Next I'll talk about the other messaging apps and briefly list the pros and cons between them and Telegram (compared to this project).

2.2.1 Whatsapp[4]

WhatsApp (formally WhatsApp Messenger) is an instant messaging computer application. The service requires users to provide a standard mobile phone number. Users can exchange text messages, images, video and audio files, as well as location information, documents, and contact information between two people or in groups. Users can communicate with others individually or in groups of individual users.

✓ Most used messaging app.

✗ No support for bots and developers.

2.2.2 Slack[5]

Slack is software that falls into the category of business collaboration tools used to instantly send messages to team members. One of Slack's features is the ability to organize team communication through specific channels, channels that can be accessible to the whole team or just to some members. It is also possible to communicate with the team through private individual chats or chats with two or more members. By integrating with different applications, software performance and team productivity can be increased. Within the platform you can in fact use Google Drive, Trello, GitHub, Google Calendar and other popular applications.

✓ Support for bots and developers.

✗ Being able to place nested replies to messages, managing automatic replies and the database is very complicated.

2.2.3 Discord[6]

Discord is a VoIP application designed for gamer communities. The Discord application was designed to be used while gaming, and has advantages such as low latency and high stability. Discord developers have added video calling and screen sharing.

✓ Support for bots and developers.

✗ The latency of messages on smartphones makes everything tedious.

2.2.4 Messenger[7]

Facebook Messenger (commonly known as Messenger) is an instant messaging or videochat application and platform developed as Facebook Chat. Users can send messages and exchange photos, videos, stickers, audio files, react to user messages, and interact with bots. The service also supports voice and video calls. Stand-alone apps support the use of multiple accounts, end-to-end encrypted conversations, and games.

- ✓ Support for bots and developers.
- ✗ The link with the facebook profile may not please many.

Chapter 3

BotFather

The Bot BotFather is a bot created by the developers of Telegram itself. It is used to support developers who want to program a bot or game for the platform. I will not explain all the commands because there are really dozens of them but I will talk about the commands mainly useful for managing bots:

- **/newbot**: this command allows you to create a new bot, first BotFather asks the user what name to give the new bot. Subsequently, the bot asks the user to select a username for the bot, this username must end with the string 'bot' and is a unique name in the world and will be the username with which the bot can be called with the syntax @username (if you select an existing name the bot will ask the user for a new one). After these steps BotFather will return the bot token.
- **/mybots**: this command is used to modify, delete or show the information of a bot, first the bot after sending the command to ask through an InlineKeyboard which bot you want to select, once selected it shows you some information and shows you a second InlineKeyboard with some commands to act on the bot itself.
- **/token**: this command makes you choose a bot via a ReplyKeyboard and gives you the token to use for the bot.
- **/revoke**: this command makes you choose a bot via a ReplyKeyboard and deletes the old token and returns you a new one, the old token is no longer usable.
- **/deletebot**: this command allows you to delete a bot, the bot is selectable via a ReplyMarkup and after selecting it BotFather asks for confirmation by making the user type the phrase 'Yes, I am totally sure.'.

N.B. It allows you to create a maximum of 20 bots per account.

Chapter 4

Telepot

4.1 Brief description of the library[8]

Telepot is the library for creating and managing Telegram bots. It is used on both nodejs and python. There are 2 versions of the library:

- **Traditional version** works on Python 2.7 and Python 3. It uses synchronous HTTP request.
- **Async version** works on Python 3.5 or above. It uses asynchronous HTTP request.

For simplicity of use and complexity of the project I opted for the traditional version.
Now I will show you the simple code of a bot to show you some basic library functions:

```
1  import telepot
2
3  def message(msg):
4      return None
5
6  def query(msg):
7      return None
8
9  bot=telepot.Bot("TOKEN")
10 bot.message_loop({'chat':message,'callback_query':query})
11
12 while True:
13         time.sleep(10)
```

The line import telepot import the library.
The token is a unique code that identifies the bot, with the command telepot.Bot(token) you associate the respective token with the designated bot. The command returns the real bot.
The bot.message_loop('chat' : message , 'callback_query' : query) command matches the various event handlers to the events themselves:

- **The chat event**: receives the function that manages the event of the text message sent by the user or of the response via ReplyKeyboard.

- **The callback_query event**: receives the function that handles the response event via InlineKeyboard.

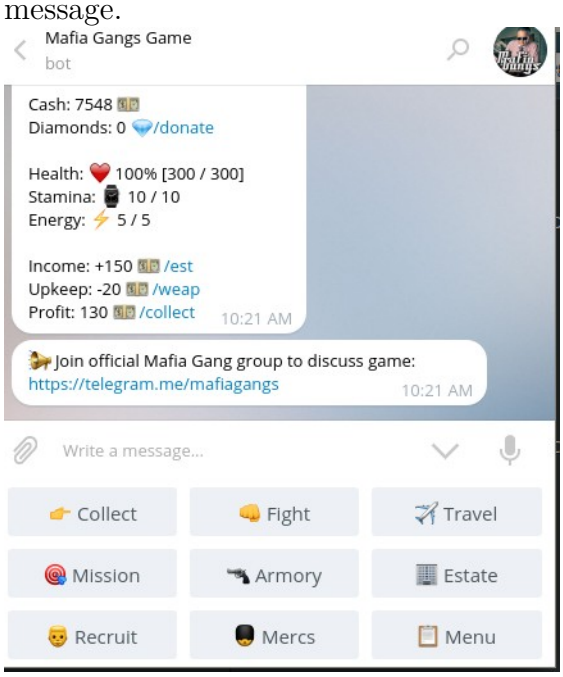
4.2 Message text

A normal text message created by the keyboard.

[9]

4.3 ReplyKeyboard

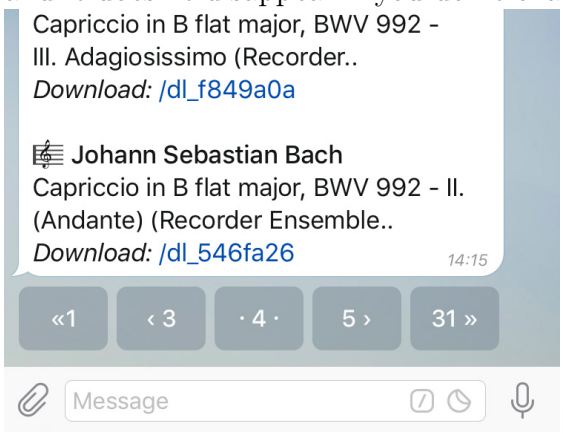
A keyboard that appears in place of the text to be entered. It can have one at a time and it can also disappear after pressing a key. The input is sent as a text message.



[10]

4.4 InlineKeyboard

A keyboard that appears below the sent message. You can have one per message and it doesn't disappear if you don't change the message.



[11]

4.5 Event handler

There are differences between the 2 types of events, even only between the messages exchanged. Below we will analyze the main differences between the 2:

4.5.1 The chat event

Now I will only analyze the text message because it is the type of message I have used most in the bot (The message passed to the chat event handler changes based on the type of message sent by the user).

```
message={'message_id': 1530, 'from': {'id': 297895076, 'is_bot':
    False, 'first_name': 'Federico Silvio', 'last_name': 'Gorrino', '
    username': 'Shawarma96', 'language_code': 'it'}, 'chat': {'id':
    297895076, 'first_name': 'Federico Silvio', 'last_name': 'Gorrino'
    , 'username': 'Shawarma96', 'type': 'private'}, 'date':
    1599752136, 'text': 'Ciao'}
```

Points of interest:

- **message["message_id"]**: Indicates the number of the message in the chat, it is used if the bot in the future wants to modify a past message.
- **message["from"]["id"]** and **message["chat"]["id"]**: The from_id indicates the user ID and the chat_id indicates the chat ID. They are not always the same, they are the same if the chat is a private chat.

```
if message["chat"]["type"]=="private":
    print("message['from']['id'] and message['chat']['id'] are
    equal")
else:
    print("message['from']['id'] and message['chat']['id'] are
    not equal")

```

- **message["text"]**: It is present only in text messages (each type of message has a different field), if present it contains the text of the message.

4.5.2 The callback_query event

On the other hand, if a user presses the key of an InlineKeyboard to the program, another type of message will arrive. Now I will analyze it and report it below.

```
message={'id': '1279449611779077718', 'from': {'id': 297895076, '
    is_bot': False, 'first_name': 'Federico Silvio', 'last_name': '
    Gorrino', 'username': 'Shawarma96', 'language_code': 'it'}, '
    message': {'message_id': 1533, 'from': {'id': 1273983096, 'is_bot'
    : True, 'first_name': 'Federicobot', 'username': '
    FedericoSilviobot'}, 'chat': {'id': 297895076, 'first_name': '
    Federico Silvio', 'last_name': 'Gorrino', 'username': 'Shawarma96'
    , 'type': 'private'}, 'date': 1599752245, 'text': 'Selezionare un
    comando per usarlo', 'reply_markup': {'inline_keyboard': [[{'text'
    : '/list', 'callback_data': 'l'}], [{'text': '/question', '
    callback_data': 'q'}], [{'text': '/report', 'callback_data': 'r'
    }], [{'text': '/start', 'callback_data': 's'}], [{'text': '/
    revision', 'callback_data': 'rv'}], [{'text': '/change_lang', '
    callback_data': 'cl'}], [{'text': '/delete_subscription', '
    callback_data': 'ds'}]]}}, 'chat_instance': '7297231428284009974',
     'data': 's'}
```

Points of interest:

- **message["message"]**: This field displays a message similar to that of the chat event but with some differences:
  - **message["message"]["from"]**: Here, instead of the user info, the info of the bot itself (the one who created the keyboard) is shown. User info is shown in the **message["from"]** field.
  - **message["message"]["text"]**: It is present only in text messages (each type of message has a different field), if present it contains the text of the message connected to the InlineKeyboard.
  - **message["message"]["reply_markup"]**: Carries a copy of the InlineKeyboard to which the button was pressed.
- **message["data"]**: Contains the code linked to the InlineKeyboard button.

4.5.3 Telepot.glance function

An honorable mention goes to the telepot.glance function. Function that receives the message and has different behavior based on whether it was called in the chat handler or the callback_query handler:

- **Chat handler**:

```
def glance_message(message):
    return get_type(message), message['type'], message['chat']['
    id']
```

```
3
```

The possible types of the message are:

```
all_content_types = [
    'text', 'audio', 'document', 'game', 'photo', 'sticker', '
    video', 'voice', 'video_note',
    'contact', 'location', 'venue', 'new_chat_member', '
    left_chat_member', 'new_chat_title',
    'new_chat_photo', 'delete_chat_photo', 'group_chat_created',
    'supergroup_chat_created',
    'channel_chat_created', 'migrate_to_chat_id', '
    migrate_from_chat_id', 'pinned_message',
    'new_chat_members', 'invoice', 'successful_payment'
]

```

- **Callback_query handler**:

```
def glance_query(message):
    return message['id'], message['from']['id'], message['data']

```

4.6 Send functions[8]

Since there are different types of data that can be sent, there are different functions to send them (one of each type). Below we will see the sendMessage (the function I used the most) and a summary of the main others.
All of these functions except sendChatAction return the saved message.

4.6.1 SendMessage

The sendMessage function is used to send text messages.

```
sendMessage(chat_id, text, parse_mode=None, disable_web_page_preview=
    None, disable_notification=None, reply_to_message_id=None,
    reply_markup=None)
```

As you can see there are several parameters, the ones I use (which I think are the most important) are the following:

- **chat_id**: the ID identifier of the chat where the program wants to send the message.
- **text**: the text of the message.
- **reply_markup**: by default it is set to None but, if different, it allows the message to set a keyboard and its type (InlineKeyboard, ReplyKeyboard, etc ...).

The return value of this function is the same as the chat_event message but with the **message["from"]** field the information of the bot itself. If the message has a keyboard, the **message["reply_markup"]** field also exists and contains a copy of it.

4.6.2 Other functions

Below I will explain the other sending functions. If the program has to send some files the maximum managed size is 50MB.

- **sendPhoto**: the function is used to send photos.
- **sendAudio**: the function is used to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .MP3 or .M4A format.
- **sendDocument**: the function is used to send general files.
- **sendVideo**: the function is used to send video files, Telegram clients support mp4 videos (other formats may be sent as Document).
- **sendVoice**: the function is used to send audio files, if you want Telegram clients to display the file as a playable voice message. For this to work, your audio must be in an .OGG file encoded with OPUS (other formats may be sent as Audio or Document).
- **sendVideoNote**: the function is used to send video messages. Video messages are rounded square mp4 videos of up to 1 minute long.
- **sendMediaGroup**: the function is used to send a group of photos or videos as an album.
- **sendLocation**: the function is used to send point on the map.
- **sendVenue**: the function is used to send information about a venue.
- **sendContact**: the function is used to send phone contacts.

- **sendGame**: the function is used to send a game.
- **sendInvoice**: the function is used to send invoices.
- **sendChatAction**: the function is used when you need to tell the user that something is happening on the bot's side. The status is set for 5 seconds or less (when a message arrives from your bot, Telegram clients clear its typing status). Returns True on success.
- **sendSticker**: the function is used to send static .WEBP or animated .TGS stickers.

4.7 GetChat function

Given the user ID, the getChat function returns information related to the user in the following format:

```
user={'id': 297895076, 'first_name': 'Federico Silvio', 'last_name':
    'Gorrino', 'username': 'Shawarma96', 'type': 'private', 'photo': {
    'small_file_id': '
    AQADBAAD7KcxG6SEwREACFbVZiNdAAMCAAOkhMERAASbJAIX44DbvUI4AwABGwQ',
    'small_file_unique_id': 'AQADVtVmI10AA0I4AwAB', 'big_file_id': '
    AQADBAAD7KcxG6SEwREACFbVZiNdAAMDAAOkhMERAASbJAIX44DbvUQ4AwABGwQ',
    'big_file_unique_id': 'AQADVtVmI10AA0Q4AwAB'}}
```

4.8 Exception

The telepot library also provides many exceptions that handle the major failures of bot man communication. They are imported with the telepot.exception library and the source code of the library can be found on the following page.[12] To be sure, I caught some of these exceptions every time a new message is modified or sent because in the beta testing phase some user behaviors have triggered them.

Chapter 5

Spacy and Natural Language Processing

5.1 Semantic similarity[13]

According to the definition of Wikipedia semantic similarity is a metric defined on a set of documents or terms, where the idea of distance between them is based on the similarity of their meaning or semantic content as opposed to the similarity that can be estimated regarding their syntactic representation (for example, the string format). These are mathematical tools used to estimate the strength of the semantic relationship between language units, concepts or instances, through a numerical description obtained according to the comparison of information supporting their meaning or describing their nature. The term semantic similarity is often confused with semantic kinship. Semantic relativity includes no relationship between two terms, while semantic similarity includes only "is a" relations. For example, "banana" is similar to "apple", but it is also related to "eat" and "cook".

5.2 Python

Python is one of the best languages for calculating semantic similarity because it is heavily stocked with libraries for machine learning and linguistic similarity is a sub-branch of it. Then the fact that python is a high-level programming language makes it much easier to write a program.

5.3 Spacy[14]

Spacy is an open source library for natural language processing, written in Python and Cython. The library is licensed under the MIT license and currently implements statistical models of neural networks in different languages; it also offers NER and tokenization capabilities for several other languages. Unlike the NLTK suite, which is widely used in research and education, spaCy is particularly suited to the creation of software applications intended for production. Starting with version 1.0, spaCy supports deep learning based analytics, allowing you to employ trained statistical models using machine learning libraries such as TensorFlow, Keras, Scikit-learn and PyTorch. In addition, spaCy's machine learning library, called Thinc, is available as an open source library for Python.

5.4 Similarity matching with spacy

Below we will look at the main functions I used for semantic similarity and how they work.

5.4.1 Load a language

To use a language in spacy you must load a language model, there is both a multilingual model and a monolingual model (for each language implemented). Models load with the following instructions.

```
import spacy
nlp = spacy.load('en_core_web_lg')
```

In the example above we have loaded a model of the English language. There are several types of models:

- **sm model**: it is a small model (<50MB). Its main feature is that it doesn't have a word vector.
- **md model**: it is a medium size model (<100MB). Start having a word vector.
- **lg model**: it is a large model (<1000MB). It has the largest word vector of all models.

Word vector[15]

Spacy calculates similarity by comparing word vectors or multidimensional representations of a word's meaning. Word vectors can be generated using an algorithm

like word2vec and usually look like this(this table is provided as an example on the official Spacy website):

```
array([2.02280000e-01,  -7.66180009e-02,   3.70319992e-01,
        3.28450017e-02,  -4.19569999e-01,   7.20689967e-02,
       -3.74760002e-01,   5.74599989e-02,  -1.24009997e-02,
        5.29489994e-01,  -5.23800015e-01,  -1.97710007e-01,
       -3.41470003e-01,   5.33169985e-01,  -2.53309999e-02,
        1.73800007e-01,   1.67720005e-01,   8.39839995e-01,
        5.51070012e-02,   1.05470002e-01,   3.78719985e-01,
        2.42750004e-01,   1.47449998e-02,   5.59509993e-01,
        1.25210002e-01,  -6.75960004e-01,   3.58420014e-01,
        # ... and so on ...
        3.66849989e-01,   2.52470002e-03,  -6.40089989e-01,
       -2.97650009e-01,   7.89430022e-01,   3.31680000e-01,
       -1.19659996e+00,  -4.71559986e-02,   5.31750023e-01], dtype=
    float32)
```

Spacy tries to attribute vectors of numerical values to objects (strings in our case). In our program, after dividing the string into tokens (words), it tries to attribute vectors of numbers to them (both to commonly used words such as "house" or "chair" and to words not present in the dictionary, in the case of words not present the returned carrier will be made of 0). After transforming the strings in this way, the similarity() method must be used, this method returns a value based on their similarity (this value is between 0 and 1, the more other the sentences are similar). The efficiency of this method is completely dependent on the size and richness of the word vector, the larger it is the more the vectors will be assigned correctly and the semantic similarity calculated efficiently.

Load a word vector

If you are using a model without word vector and you have a file with the word vector you can upload it through the following instructions:

```
import spacy
from gensim.models import Word2Vec
nlp = spacy.load('it_core_news_sm')
model = Word2Vec.load('wiki_iter=5_algorithm=skipgram_window=10_size
    =300_neg-samples=10.m')
keys = []
for idx in range(733392):
    keys.append(model.wv.index2word[idx])
nlp.vocab.vectors = spacy.vocab.Vectors(data=model.wv.vectors, keys=
    keys)
```

5.4.2 Remove stopwords, punctuation, and pronouns

If you want to calculate the semantic similarity between 2 sentences, you must definitely preprocess the sentences. To do this, you need to remove 3 types of words from them:

- **stop words**: they are the most used words in the language, precisely because of their high frequency they do not have much relevance in semantic similarity (the, are, a, etc...).
- **punctuation marks**: (.,„,?,!,etc...).
- **pronouns**: (i, he, she, we, they, who, whoever, etc...).

We can remove these elements from the sentence with the following code:

```
def process_text(text):
    doc = nlp(text.lower())
    result = []
    for token in doc:
        if token.text in nlp.Defaults.stop_words:
            continue
        if token.is_punct:
            continue
        if token.lemma_ == '-PRON-':
            continue
        result.append(token.lemma_)
    return " ".join(result)
```

The **doc = nlp(text.lower())** command is used to transform all text into lowercase and then divide it into words, doc is a vector of words in a lower case.

5.4.3 Calculate similarity

To calculate the similarity between 2 sentences you need to preprocess them both and then use the **similarity** library function.

```
def calculate_similarity(text1, text2):
    base = nlp(process_text(text1))
    compare = nlp(process_text(text2))
    return base.similarity(compare)
```

The function written above returns a number between 0 and 1 which indicates the degree of similarity between the sentences. The closer the number is to 1, the more similar the sentences are, the closer it is to 0 the more different the sentences are.

5.4.4 Is an element in a vector?

In my program it happened to me several times that I had to calculate the semantic similarity between a sentence and all the elements of an array, to do this I used the following function:

```
def match_array(self,txt,lang,vett):
    e=''
    val=0
    for elem in vett:
        num=calculate_similarity(txt,elem,lang)
        if num > val:
            val=num
            e=elem
    if val>0.8:
        return e
    return None
```

The above function calculates the maximum similarity between a sentence and each element of an array. For example, if we have the phrase "hi, I'm a student" and the array ["I eat", "hi, I'm a professor", "the student watches TV"] the algorithm works as follows:

- set the variable val to 0
- calculate the similarity between "hi, I'm a student" and "I eat" (for example 0.1)
- compares val with the new value, if the new value is greater than val updates val (0.1 is greater than 0 so val becomes equal to 0.1)
- calculate the similarity between "hi, I'm a student" and "hi, I'm a professor" (for example 0.9)
- compares val with the new value, if the new value is greater than val updates val (0.9 is greater than 0.1 so val becomes equal to 0.9)
- calculate the similarity between "hi, I'm a student" and "the student watches TV" (for example 0.3)
- compares val with the new value, if the new value is greater than val updates val (0.3 is less than 0.9 so the value of val does not change)

In the field of machine learning it is essential to define a threshold value, a value that if exceeded says that two elements can be defined as similar. In this case I define the threshold value at 0.8, if the value of val is greater than it I return the

element that generated it otherwise I return None (in the example above I would return "hi, I'm a professor").

5.5 Googletrans library

In the project sometimes I also needed to translate a sentence from one language to another, for this purpose I used the google translator library.

5.5.1 Translate a string

In the following piece of code we can see an example of how it is possible to translate a sentence (**string**) from a source language (**src**) to a target language (**dest**), the code returns the translated sentence in the new language.

```
1 def translate(self,string,src,dest):
2     trans=Translator()
3     t=trans.translate(string,src=src,dest=dest)
4     return t.text
```

Chapter 6

Chatbots for Education and Learning

Mine is not the first application that uses chat bots for educational purposes. While researching, I came across the ChatCompose site that deals with this topic describing the main uses of bots in today's education.

6.1 Not all students are equals[16]

In my school experience it has happened to me several times to see the difference in knowledge between people of the same class and the article quoted here starts with similar considerations. He continues talking about tutors for people who are left behind unfortunately also exposes the fact that it is an expensive solution and certainly Chatbots are a cheaper alternative. The purpose of these Chatbots is to be able to give basic advice and to adapt to the needs of the student. But going forward, we always try to develop more and more complex bots in such a way as to be able to replace a tutor in all aspects.

6.2 Advantages of using a ChatBot[16]

Linguistics is a very complex topic and at times I have come across bots that do not understand what the user is going to do so I fully agree with the ChatCompose article when it says that first of all it is essential that the bot understand the user's intentions.
Once the user's will has been identified, according to the article the bot can respond to the user in 4 different ways:

- A generic and predefined text response.

- A context using data that the user provides.
- Data stored in databases.
- A specific action (the chatbot interacts with one or more applications).

The article also illustrates several advantages for female friends who adopt chatbots:

- Reducing costs of personnel that provided customer service.
- Immediate support and responses for users.
- Better user experience.
- Possibility of learning about preferences of the user when using artificial intelligence systems.
- Ease of integration in different digital platforms, without the need to download applications or additional programs.

6.3 Applications for chatbots in education[16]

The article goes on to say that teacher communication is now being automated and takes the example of the GoogleClassroom or forum tool to bring the learning experience beyond the walls of the classroom, it also illustrates how much these methods help students to learn. more and to perform better. With the use of bots, the interaction on the teacher side could be partially cut and the student would have an expert on the subject at his disposal 24 hours a day ready to answer any question. Bill Gates Jr. also expressed himself on the subject by saying that in his career as a student the comparison with experts was fundamental and the bot is ultimately an expert program in that subject. So the founder of Microsoft was very positive about these interactions. The article continues by exposing some innovations of Chatbots in the communication between the student and the teacher:

- **It detects the emotional state of the students**: if the bot detects an altered emotional state on the part of the student, it can modify its responses based on it.
- **Provides personalized learning**: the bot can provide more targeted learning for each student by diversifying the paths of the members of a class.
- **It allows the teacher to reduce time**: the bot responds immediately does not have to wait for human intervention and therefore the professor saves time in these interactions.

Ultimately, the article advises professors to use the time saved in research and improve the student experience. Some main advantages of this project are certainly:

- **Store and analyze data effectively**: artificial intelligence can analyze the data collected by the bot to improve the student experience and improve the course accordingly.
- **Improves access to education**: not having a physical location, the bot can help students around the world indiscriminately by allowing a higher degree of literacy.

Chapter 7

The classes in Python

7.1 Basic elements of a class

Python classes are created with the following syntax:

```
...
class nameClass (baseClass1, baseClass2, ..., baseClass(k-1),
    baseClassk):

    def nameFunc1(self,...):
        ...

    def nameFunc2(self,...):
        ...

    ...

    def nameFunc(k-1)(self,...):
        ...

    def nameFunck(self,...):
        ...
```

Now I'll explain the various main elements of a Python class:

- **nameClass**: is the name of the class.
- **baseClass**: are the class or interfaces from which the class inherits.
- **nameFunc**: are the names of the functions, the functions usually take as their first parameter the instance of the class that called the function.
- **self**: it is the instance of the class that called the function.

7.2 Basic functions of a class[17]

The main functions of a class in python:

- **___new___**: this is a astatic method which takes the calling class as its first argument and the other arguments defined by the programmer (___new(cls, ...)___). It is the first function called when constructing an object and returns the object instance (new if it is a normal class, old if it is a Singleton class). We can see below an example:

```
...
class Obj1:

    def __new__(cls,var1,var2,...,var(k-1),vark):
        ...
        return istance

...
obj=Obj1(var1,var2,...,var(k-1),vark)
...
```

- **___init___**: this method is the second method that is called at initialization, it is called inside the ___new___ method between instantiation and return. It receives the instance as its first parameter and as other parameters those defined by the programmer. It is used to initialize the variables when the instance is created (the variables of the instance can be used by the other methods of the class) and always returns None. Now I write an example below:

```
...
class Obj1:

    def __new__(cls,var1,var2,...,var(k-1),vark):
        ...
        istance.__init__(val1,val2,...,val(k-1),valk)
        ...
        return istance

    def __init__(self,val1,val2,...,val(k-1),valk):
        self.val1=val1
        self.val2=val2
        ...
        self.val(k-1)=val(k-1)
        self.valk=valk

```

```
...
obj=Obj1(var1,var2,...,var(k-1),vark)
...
```

N.B. The ___new___ method can also be omitted, the following syntax will be used in case (in this case the new method is inherited from the parent).

```
...
class Obj1:

    def __init__(self,val1,val2,...,val(k-1),valk):
        self.val1=val1
        self.val2=val2
        ...
        self.val(k-1)=val(k-1)
        self.valk=valk

...
obj=Obj1(val1,val2,...,val(k-1),valk)
...
```

- **___del___**: this method is the destructor of the class, it is used when an instance is destroyed to facilitate memory freeing and to perform operations during destructions. Python already deals with a basic method of destroying the object but in some cases it is a good idea to override it.
- **The comparison functions**: They are all functions of the type nameFunc (self, other) and are the following:
  - **___lt___**: implement self<other comparison.
  - **___le___**: implement self<=other comparison.
  - **___eq___**: implement self==other comparison.
  - **___ne___**: implement self!=other comparison.
  - **___gt___**: implement self>other comparison.
  - **___ge___**: implement self>=other comparison.

Chapter 8

The bot I developed

8.1 The classes

For this project I have developed several classes that can be divided into 2 categories: the bot classes and the non-bot classes.

8.1.1 The bot classes

Bot classes are the classes that manage telegram bots. They all derive from the **Bot class** which can be found in the **bot_class.py** file.

```
1  import telepot
2
3  class Bot:
4
5      def __init__(self,token,message=lambda msg : None,query=lambda
       msg : None):
6          self.token=token
7          self.bot_instance=telepot.Bot(token)
8          self.bot_instance.message_loop({'chat':message,'
       callback_query':query})
9
10     def get_bot(self):
11         return self.bot_instance
12
13     def get_token(self):
14         return self.token
```

The bot classes are:

- **BotAdmin**: is a bot that receives bug reports from students and professors.

- **BotCreation**: is a bot that allows you to create new bots for students and new credentials for teachers.
- **BotGetlink**: is a bot where students can find all links to bots dedicated to them.
- **BotPwd**: is a bot where you have a log of all the passwords of the bots and of those who request them.
- **BotStudent**: it is the only bot with multiple instances. There is a bot for each subject and it is used by students to interact with the system.
- **BotTeacher**: requires login. On the basis of the credentials entered, the professor can access the subject. It allows the teacher to interact with the system.

All bots except **BotStudent** are managed by Singleton classes. Below is an example of a Singleton:

```
class OnlyOne(object):

    class __OnlyOne:

        def __init__(self):
            self.val = None

    instance = None

    def __new__(cls):
        if not OnlyOne.instance:
            OnlyOne.instance = OnlyOne.__OnlyOne()
        return OnlyOne.instance
```

Whenever a Singleton class is instantiated, the __new__ method is called:

- The first time this operation is done, the instance variable is equal to None, so a new instance of the class is created.
- In other cases the instance variable is different from None, so the instance of the class already present in the instance variable is returned.

8.1.2 The BotId class

The BotId class is a Singleton class instantiated into 3 bot classes: **BotCreation**, **BotStudent** and **BotTeacher**.

The peculiarity of these classes is the fact that they have to save states on the server to maintain their workflow. Unfortunately, keeping too many states on the server can be dangerous because the ram can overload, that's why the **BotId** class was created. This class closes old connections in case too much time passes, the user requests a new connection, or the maximum connection limit has been exceeded. This class has 3 main functions with which it interfaces with the outside world:

- The **add_time_id** function: this function adds a new id. This function is called when a new command is executed.

```
def add_time_id(self,chat_type,lang_class,lang,from_id,chat_id,
    val,name):
     self.add_id(from_id,chat_id,val,name)
     self.set_time(from_id,chat_id,name)
     self.delete_old_ids(chat_type,lang_class,lang)
```

- The **check_time_id** function: this function returns an id if it exists, otherwise it returns 0. This function is called when an old command requires an input to go forward (so by the id the calling command is recognized and the next action is performed).

```
def check_time_id(self,chat_type,lang_class,lang,from_id,chat_id,
    name):
     num=self.check_id(from_id,chat_id,name)
     self.set_time(from_id,chat_id,name)
     self.delete_old_ids(chat_type,lang_class,lang)
     return num
```

- The **del_time_id** function: this function returns delete an old id. This function is called when a command completely ends its life cycle.

```
def del_time_id(self,chat_type,lang_class,lang,from_id,chat_id,
    name):
     self.del_id(from_id,chat_id,name)
     self.set_time(from_id,chat_id,name)
     self.delete_old_ids(chat_type,lang_class,lang)
```

The id_times and id_commands arrays

These 2 arrays are the main variables of this class. They both handle items in two ways based on chat_type:

- **chat_type=="private" (from_id!=chat_id)**: When a request comes from a private chat, the following arrays are of the form **array[name][chat_id][from_id]**
- **chat_type!="private" (from_id==chat_id)**: When a request arrives from a non-private chat, the following arrays are of the form **array[name][chat_id]**

But what are the differences between the 2 arrays? The above functions always save a datetime in id_times array when a command is sent but instead have a different behavior with the id_commands array **(in add_time_id the id is added, in check_time_id the id is returned and in del_time_id the id is deleted)**.

The name variable

In the previous paragraphs we saw that an index of the above vectors is the variable name but what is inside it? The variable name is a string that contains an identified value of the calling bot:

- **BotStudent**: if the function is called by this bot the variable name contains the name of the subject.
- **BotTeacher**: if the function is called by this bot the variable name contains the string "teacher".
- **BotCreation**: if the function is called by this bot the variable name contains the string "creation".

8.1.3 UserBanned class

This class keeps track of users banned from BotCreation and BotTeacher (since it is a Singleton class the same instance is saved in BotTeacher and in BotCreation, so the users banned by the 2 bots are the same).
The whole class is based on an array of counters, the elements of the array are always of the form array [chat_id].
Also this class interfaces outside with 3 methods very similar to the previous ones:

- The **add_ban** function: this function increments the counter based on the chat_id, if there is no counter for that chat_id it initializes it to 1. This function is called whenever a user makes a mistake in typing a password.

```
1 def add_ban(self,chat_id):
2     if chat_id in self.banned_user:
3         self.banned_user[chat_id]+=1
```

```
    else:
        self.banned_user[chat_id]=1
    self.write_ban()
```

- The **check_ban** function: this function looks to see if that chat_id's counter is greater than 99, if so the user is not allowed to continue. This function is called at the beginning of the chat event and query_callback event handles.

```
def check_ban(self,chat_id):
    if chat_id in self.banned_user and self.banned_user[chat_id
    ]>99:
        return True
    return False
```

- The **del_ban** function: this function clears, if it exists, the counter linked to chat_id. This function is called if the user enters the correct password.

```
def del_ban(self,chat_id):
    if chat_id in self.banned_user:
        del self.banned_user[chat_id]
        self.write_ban()
```

8.1.4 The non-bot classes

Non-bot classes are delegated tasks that do not have to interface with the user.

- **Database**: to this class are delegated all the functions to interface with the database.
- **Language**: to this class are delegated all the functions concerning the matching between sentences and the translation of sentences.
- **Node**: there are multiple instances of this class, each handling all subject information that is not relevant to the bot itself.
- **Tree**: this class splits requests made by other bots to requested student bot.

The non-bot classes are all Singletons except the Node class.

Chapter 9

How the program starts and in what order the classes are started

To initialize the bot just start the script that is in init.py, I report it below explaining all the steps it takes:

```
import time
from tree_class import Tree
from database_class import Database
from bot_admin_class import BotAdmin
from bot_creation_class import BotCreation
from bot_getlink_class import BotGetlink
from bot_pwd_class import BotPwd
from bot_teacher_class import BotTeacher

database=Database()
bot_admin=BotAdmin()
bot_pwd=BotPwd()
database.set_bot_pwd(bot_pwd)
database.set_bot_admin(bot_admin)
tree=Tree()
bot_creation=BotCreation()
bot_getlink=BotGetlink()
teacher=database.get_teacher()
bot_teacher=BotTeacher()
database.set_bot_teacher(bot_teacher)

print("Init complete")

```

```
while True:
    time.sleep(10)
```

In the first section we import the time library plus some Singleton classes I created. The second section initializes all classes and saves instances of several bots in the instance of the Database class. As you can see, neither the BotStudent class, the Node class and the Language class are instantiated. Below we see what the Tree class looks like in the ___init___ function that instantiates various instances of the BotStudent class.

```
...
class Tree:
    class Singleton:

        def __init__(self):
            self.lang=Language()
            self.array={}
            self.num=0
            self.database=Database()
            data=self.database.get('/bots/students','')
            for topic in data:
                self.array[topic]=BotStudent(topic)
...
```

As you can see, the Tree class has an associative array that saves the corresponding bot for each topic name. If any bot has to send a message to a BotStudent it will have an instance of the Tree class and will select the bot by the topic name. Instead the BotStudent if it has to communicate with another bot asks the Database class to pass it an instance of the desired bot.
Each instance of the BotStudent class when initialized creates its own instance of the Node class. The BotStudent classes have a one-to-one relationship with the Node classes, these classes divide the data to be stored. Below I will summarize in a table the variables that you save the BotStudent class and those that you save the Node class.
The variables in the BotStudent class are used for:

- **keyboard**: this variable contains the InlineKeyboard with the commands available to the student.
- **students**: this associative array contains all students enrolled in the course divided by language.
- **teachers**: this associative array contains all teachers enrolled in the course divided by language.

BotStudent	Node
keyboard	name
students	database
teachers	json_array
collaborators	language
node	questions
banned_user	hash
singleton	

- **collaborators**: this associative array contains all collaborators enrolled in the course divided by language.
- **node**: contains an instance of the Node class (linked to the topic).
- **banned_user**: contains a list of users banned from the topic.
- **singleton**: contains a copy of the instance of the BotId class.

The variables in the Node class are used for:

- **name**: the name of the topic.
- **database**: a copy of the instance of the Database class.
- **json_array**: an associative array that contains the subject questions divided by language (with a syntax very similar to that saved in the database).
- **language**: a copy of the instance of the Language class.
- **questions**: if the command needs to save data on the current application it gives opera while waiting for another output it saves them in this associative array.
- **hash**: contains the hash of the password related to the topic.

Chapter 10

Firebase

10.1 Brief description of the software[18]

Firebase is a platform developed by Google that allows you to have an online database and other useful tools. It is the platform where I saved the entire project database and it is the external site with which the Database class interfaces.

10.2 The realtime database[19]

This database is an online json file that the program can make calls to. To make a call you must know the database link.

```
from firebase import firebase

db = firebase.FirebaseApplication(link,authemtication)
```

A second parameter can be entered if authentication is required, but it is usually set to None (authentication = None). Authentication for Firebase is nothing other than creating a token according to the JWT standard and inserting it in place of the None mentioned above. The database can limit write and read permissions. You can also contact the database by means of asynchronous calls by creating a pool of processes, this pool remains alive until the death of the main process and returns the various data of the database.
There are 3 possible types of queries:

- **Get function**: it is used to get the data from the database.

```
data=db.get(path,name)
```

```
2
```

- **Put function**: it is used to put data on the database.

```
1 data=db.put(path, name=name, data=data)
2
```

- **Delete function**: it is used to delete data from the database.

```
1 data=db.delete(path, name)
2
```

All 3 functions return the data modified after the modification (delete always returns None). The path is a variable of the string type and in the format: "/name1/name2/.../name(k-1)/namek". Indicates the json path requested by the user. For example if on the server we have the following json:

```
1  {
2      "val1":
3      {
4          "val1_1":"good",
5          "val1_2":"bad"
6      },
7      "val2":"hey",
8      "val3":
9      {
10         "val3_1":"middle"
11     }
12 }
```

and the path used in a get function matches "val1" the returned value will be {"val1_1": "good", "val1_2": "bad"}.
The variable name is used to lengthen the path by one level, taking the previous json, if we have a path equal to "val1" and a name equal to "val1_1" the returned value will be "good".
The put function also uses the data parameter which represents the data to load into the database. Returning to the previous example, if in a put function we have path = "val3", name = "val3_2" and data = {"val3_2_1" = "cabodi", "val3_2_1" = "camurati"} the function returns {"val3_2_1" = "cabodi", "val3_2_1" = "camurati"} and on the online database you will have the following json:

```
{
    "val1":
    {
        "val1_1":"good",
        "val1_2":"bad"
    },
    "val2":"hey",
    "val3":
    {
        "val3_1":"middle",
        "val3_2":
        {
            "val3_2_1":"cabodi",
            "val3_2_2":"camurati"
        }
    }
}
```

10.3 The subdivision of my Json

My Json is immediately divided into 2 parts:

- **The bots part**: this part contains all the information needed to make the bots work.
- **The translate part**: this part contains supported languages with various strings already translated.

10.3.1 The strings "de", "en", "es", "fr" and "it"

They represent languages supported by bots:

```
{
    "de": "deutch",
    "en": "english",
    "es": "spanish",
    "fr": "french",
    "it": "italian"
}
```

The internal structure of these languages is almost the same (if "it" will have the "val" field also "fr" has it) and for non-redundancy when in the structures below there will be redundant and predictable fields I will put the ... in place of the fields.

10.3.2 The bots part

/bots/admin

The structure of this field is the one below:

```
{
    "de":
    {
        "ids":
        {
            0:chat_id1,
            1:chat_id2,
            ...
            (k-2):chat_id(k-1),
            (k-1):chat_idk
        },
        "students":
        {
            "string1":"datetime1",
            "string2":"datetime2",
            ...
            "string(k-2)":"datetime(k-1)",
            "stringk":"datetimek"
        },
        "teachers":
        {
            "string1":"datetime1",
            "string2":"datetime2",
            ...
            "string(k-2)":"datetime(k-1)",
            "stringk":"datetimek"
        }
    },
    "en":
    {
        ...
    },
    "es":
    {
        ...
    },
    "fr":
    {
        ...
    },
    "it":
    {
```

```
43          ...
44      },
45      "token":"bot_token"
46  }
```

- **The bot_token in the token field**: this variable contains the token that allows the BotAdmin to work.
- **The array of chat_id in the ids field**: if someone who has set language as [lang] wants to report bugs, the report will be sent to all ids in the array **/bots/admin/[lang]/ids**.
- **The associative array in students and teachers**: the fields of this associative array are of the "string":"datetime" type. The "string" contains the bug reported to the administrator, and the "datetime" contains the date and time it was reported. After 7 days a reported bug is deleted from the database.

/bots/creation

Contains the token of the BotCreation class.

/bots/getlink

Contains the token of the BotGetlink class.

/bots/key_id

In this section you can find only one type of path and it is of the type **/bots/key_id/[topic]**. The topic is the name of the subject concerned.
Inside this path there is a structure like:

```
1  {
2      "chat_id1":message_id1,
3      "chat_id2":message_id2,
4      ...
5      "chat_id(k-1)":message_id(k-1),
6      "chat_idk":message_idk,
7  }
```

- **The chat_id variables**: these variables contain chat_id's.
- **The language part**: these variables contain message_id's

This structure saves the identifiers of all old messages with an InlineKeyboard. If a new message with an InlineKeyboard is sent or the bot is restarted, all previous InlineKeyboards are deleted (they are found through the message_id, after the field is updated or deleted from the database).

/bots/pwd

The structure of this field is the one below:

```
{
    "admin":chat_id,
    "bot":"token",
    "requests":
    {
        "chat_id1":
        {
            0:"datetime1",
            1:"datetime2",
            ...
            (k-2):"datetime(k-1)",
            (k-1):"datetimek"
        },
        "chat_id2":
        {
            ...
        },
        ...
        "chat_id(k-1)":
        {
            ...
        },
        "chat_idk":
        {
            ...
        }
    }
}
```

- **The chat_id in the admin field**: these variables contain chat_id's.
- **The token in the bot field**: this variable contains the token identifying the PwdBot.
- **The requests field**: in this field there are all the password change requests (with date and time) divided by the chat_id that made them. I also collect the

date and time because I have set a limit of these requests (10 every month). If a request is very old firebase deletes it.

/bots/students

The structure of this field is the one below:

```
{
    "de":
    {
        "collaborators":
        {
            0:chat_id1,
            1:chat_id2,
            ...
            (k-2):chat_id(k-1),
            (k-1):chat_idk
        },
        "questions":
        {
            "question1":
            {
                "answer":"answer_question"
                "ids":
                {
                    0:chat_id1,
                    1:chat_id2,
                    ...
                    (k-2):chat_id(k-1),
                    (k-1):chat_idk
                },
                "revision":
                {
                    0:"string1",
                    1:"string2",
                    ...
                    (k-2):"string(k-1)",
                    (k-1):"stringk"
                }
            },
            "question2":
            {
                ...
            },
            ...
            "question(k-1)":
            {
```

```
                    ...
                },
                "questionk":
                {
                    ...
                }
            },
            "students":
            {
                0:chat_id1,
                1:chat_id2,
                ...
                (k-2):chat_id(k-1),
                (k-1):chat_idk
            },
            "teachers":
            {
                0:chat_id1,
                1:chat_id2,
                ...
                (k-2):chat_id(k-1),
                (k-1):chat_idk
            }
        },
        "en":
        {
            ...
        },
        "es":
        {
            ...
        },
        "fr":
        {
            ...
        },
        "hash":"hashed_pwd",
        "it":
        {
            ...
        },
        "token":"bot_token"
}
```

- **The questions associative array**: in this field all the questions and the relative data connected to them are saved. All this is saved like this:
  - **question1, question2, ..., question(k-1) and questionk**: these strings

are the questions.

  - **The answer_question in the answer field**: this string is the answer to the relevant question.
  - **The revision field**: this array contains the comments to the revisions to the relevant question sent by students.

- **The bot_token in the token field**: this variable contains the token identifying the TeacherBot.
- **The collaborators, the students and the teachers fields**: in this field there are the collaborators, the students and the teachers registered for that subject in that language.
- **The hashed_pwd in the hash field**: this field contains the hash of the password to allow professors to have the credentials to access this topic in the BotTeacher.

/bots/teachers

The structure of this field is the one below:

```
{
    "banned":
    {
        "chat_id1":num1,
        "chat_id2":num2,
        ...
        "chat_id(k-1)":num(k-1),
        "chat_idk":numk,
    },
    "token":"bot_token"
}
```

- **The bot_token in the token field**: this variable contains the token identifying the TeacherBot.
- **The banned field**: in this field there is an associative array with chat id as key and number as value, this array is the one used by the UserBanned class.

10.3.3 The translate part

In the path **/bots/translate/[lang]** there is an associative array with sentences translated into various languages, inside these strings sometimes there are strings

equal to "XXX" and "YYY", these strings will be replaced by the program with other strings according to the situation.
Below we will analyze strings and when they are used:

- **/bots/translate/[lang]/answer**: this string is used when a professor wants to answer a question, it is sent only to the teacher, the string "XXX" is replaced by the question to be answered.
- **/bots/translate/[lang]/answer_q**: this string is used when after a professor has answered the question, it is sent to both the students and the teacher, the string "XXX" is replaced by the question to be answered and the string "YYY" is replaced with the answer.
- **/bots/translate/[lang]/banned_q**: this string is used when after a professor has banned the question, it is sent to both the students and the teacher, the string "XXX" is replaced by the banned question.
- **/bots/translate/[lang]/banned_user**: this string is sent to a user when they try to operate on a bot where they have been banned.
- **/bots/translate/[lang]/bug**: this string is sent to a student or professor when they report a bug.
- **/bots/translate/[lang]/canc**: this string is used to implement the cancel key in the ReplyKeybord.
- **/bots/translate/[lang]/collaborator**: this string is sent to a professor when he registers as a collaborator.
- **/bots/translate/[lang]/command**: this string is sent along with an InlineKeyboard to enable commands.
- **/bots/translate/[lang]/comment**: this string is used to enable a button in the command "/revision" el bot student to signal to the teacher that there are no comments about it.
- **/bots/translate/[lang]/del_lang**: this string is sent to students when all teachers of a subject unsubscribe from the selected language, the string "XXX" is replaced by the language.
- **/bots/translate/[lang]/delete_s**: this string is sent to students when they unsubscribe from the bot.
- **/bots/translate/[lang]/disable**: this string is sent to students when they try to write to a bot where no professors are enrolled.

- **/bots/translate/[lang]/empty**: this string is sent when requesting access to a list with no elements.
- **/bots/translate/[lang]/error**: this string is sent when sending an illegal string.
- **/bots/translate/[lang]/error_q**: this string is used when the student tries to post a question that does not conform to the bot standards.
- **/bots/translate/[lang]/lang**: this string is sent when you submit a question that does not end with a question mark.
- **/bots/translate/[lang]/new_button**: this string is used to create the "new question" button in the ReplyKeyboard.
- **/bots/translate/[lang]/new_lang**: this string is used to notify the student that a new language has been enabled for the bot, the string "XXX" is replaced by the language.
- **/bots/translate/[lang]/new_q**: this string is sent to professors when a student submits a new question, the string "XXX" is replaced by the new question.
- **/bots/translate/[lang]/q_not_found**: this string is sent when trying to select a non-existent question (in some contexts).
- **/bots/translate/[lang]/question**: this string is sent to the student when he selects the command to send a question to the teacher.
- **/bots/translate/[lang]/report**: this string is sent to the student or professor when he selects the command to report a bug.
- **/bots/translate/[lang]/restart**: this string is sent to the student or professor when the bots are restarted.
- **/bots/translate/[lang]/revision**: this string is sent to the student when he asks for a review of the answer to a question.
- **/bots/translate/[lang]/roles**: this string is sent together with the ReplyKeyboard to select the role of the professor.
- **/bots/translate/[lang]/rv_comment**: this string is sent after the student has selected the question to ask for review and asks the student if he or she wants to comment on the choice.

- **/bots/translate/[lang]/select**: this string is sent so a user must select a question (in a ReplyKeyboard) for various purposes.
- **/bots/translate/[lang]/select_hint**: this string is sent when a professor wants to add a question through the "/add_hint" method.
- **/bots/translate/[lang]/select_q**: this string is sent before a student submits a new question, it gives the student suggestions similar to the question they typed.
- **/bots/translate/[lang]/setted_lang**: this string is sent after a user selects the bot language.
- **/bots/translate/[lang]/start**: this string is sent after the user sends the "/start" command, the string "XXX" is replaced with the topic name in the user's language.
- **/bots/translate/[lang]/teacher**: this string is sent to a professor when he registers as a teacher.
- **/bots/translate/[lang]/timeout**: this string is sent the user starts a command and does not complete it in a certain amount of time.
- **/bots/translate/[lang]/wait_q**: this string is sent to the student when the student submits a question that the professor has not yet answered, the string "XXX" is replaced by the question to be answered.
- **/bots/translate/[lang_1]/[lang_2]**: this string contains the name of [lang_2] in [lang_1].

Chapter 11

Bot commands

In my project there are 3 bots that take various commands as input, in this chapter I will explain what commands there are and how to use them.
The commands can be sent via text message or via the InlineKeyboard created by the /start command.

11.1 Private chats vs groups

Before continuing I have to talk in depth about the differences in my bot's behavior between private chats and groups. Below I am attaching 2 illustrative screens with explanations.

11.1.1 Private chats

- in the chat there is **only one bot**: therefore the commands sent by the user can only be addressed to that bot.
- in the chat there is **only one user**: therefore the replies sent by the bot can only be addressed to that user.

11.1.2 Groups

Groups are certainly a more complicated case to manage.

- in the chat there are **one or more bots**: bots do not necessarily have different commands, they could have one or more identical commands, if the command is sent with an InlineKeyboard nothing changes but if the command is sent verbatim it is a good idea to send it with the syntax /[command]@[bot_name].
- in the chat there are **one or more users**: 2 users could wait for the answer from the same command, to avoid misunderstandings I made sure that the bot in the groups tagged the recipient of the answer.

11.1.3 The match_command function

```
def match_command(command,msg,chat_type,username):
    if chat_type=="private":
        return msg==command
    else:
        return msg==(command+"@"+username)
```

This function created by me is used to recognize if a command comes from a private chat or from a group, the msg variable is the command sent by the user, the command if it is in a private chat is compared with the command variable, if not it is compared with the string command + @ + the username of the bot. The command variable is a string decided by the program, each time a message arrives this function is called for each command implemented in the bot and this variable assumes all the values of the aforementioned commands (for example "/start").

11.2 Serial vs parallel commands

There are 2 ways to manage the commands, parallel or sequential. Thanks to the structure I created and explained in chapter 6.1.2 the users of the BotStudent and the BotTeacher can send commands in parallel, the BotCreation does not need this because it is advisable to use it only in private chats.

Even if our bots act with parallel commands, let's see for a moment what the difference is between the 2 paradigms with examples. We define two commands:

- **/is_string**: when this command is launched it waits for a subsequent text message, if there are only alphabetic characters it returns true otherwise false.
- **/is_number**: when this command is launched it waits for a subsequent text message, if there are only numeric characters it returns true otherwise false.

We also assume that if a command is launched immediately after another it overwrites it if the first command has not yet been completed.
What happens in the 2 cases with the following dialogflow in a group chat?

11.2.1 Serial commands

In the case of serial commands the bot does not remember which user sent a command, so in the case above it happens that the bot sees the command /is_string, then sees the command /is_number and overwrites /is_string, then sees the string

"hello" but it has the /is_number command saved so it prints false, then sees the number 67 but no longer has any command running.

11.2.2 Parallel commands

In the case of parallel commands, the bot remembers which user sent which command. In the case shown above the bot sees the command /is_string sent by user_1 and the command /is_number sent by user_2, then it sees the string "hello" sent by user_1 and prints true and finally sees the number 67 sent by user_2 and print true.

11.3 The /start command

The /start command is the basic command and starts the bots. In the BotCreation it always shows a message with the InlineKeyboard, while in the BotStudent and in the BotTeacher it changes whether the bot has any information about the user who is using it or not.

- **if the bot has no information about the user**: in this case there is a set up to do, the student will be asked to set a main language, the teacher first the topic and then the relative password, then the teacher is asked to select a language and finally a role. The features chosen in this set up are not per user but per chat (all users in the chat have the same set up).
- **if the bot has information about the user**: in this case there is no set up to do, once the command /start is sent a message is sent with the

InlineKeyboard related to the bot.

N.B. The selectable roles for the professor are the teacher and the collaborator, they have the same duties and the same permissions, the only difference is that if you have the role of teacher you will be notified in case of a new question, of a requested review , etc ... if you are registered as a collaborator you will never receive a notification.

11.4 BotStudent

Since this bot was designed for students, I asked them what they needed. After some dialogue, the list of commands that we see below came out:

- **/list**: this command sends a txt file to the user with all topic questions he sent (with the answers if they are present).
- **/question**: this command sends a new question to the professors designated for the subject. Once the question has been typed in, a correlation is made between it and the database (see point 5.4.4) based on the value of this correlation there can be 3 cases:
  - **val==1**: if you have this value the question is certainly in the database so you send an answer to the user based on the status of the answer.
  - **0.8<=val<1**: if you have this value, the question may be in the database but it is not certain, therefore the most similar questions are shown to the user and a "New question" button in a ReplyKeyboard, once the option has been chosen, the user is notified based on the response status. If the user presses the "New Question" button the question is added to the database and the student and teachers are notified accordingly.
  - **val<0.8**: if this value is given the question is not in the database, so it is entered in the database and the student and teachers are notified.

  N.B. In all 3 cases the chat_id of the person who sent the question is saved in the database.
  There are 4 possible statuses for the response:
  - **the question is not in the database**: the question is added to the database and the student and teachers are notified.
  - **the question is in the database but there is no answer**: the bot will respond to the student by saying that the question is waiting to be answered.

  - **the question is in the database but the answer is "BANNED"**: the bot will respond to the student by saying that the question has been banned.
  - **the question is in the database but the answer is different from "BANNED"**: the bot will say that there is an answer to the question and forwards it to the student.

- **/report**: allows you to report a bug to the admin, the message seen by the admin will be of the type "students: XXX" with the reported bug in place of the XXX. This message is sent only if a similar one does not already exist in the database.

- **/revision**: this command allows you to ask teachers to review an answer, you can also add a comment to the review. If a review with such a comment does not exist it is forwarded to the teachers.

- **/change_lang**: this command allows you to change the language of the bot in the chat, you can choose between the languages enabled by the professors via a ReplyKeyboard.

- **/delete_subscription**: cancel the initial chat set up, the chat will no longer receive notifications from the bot unless an answer to a question the student is interested in is changed.

11.5 BotTeacher

Since this bot was designed for professors and the dialogue due to covid was poor due to covid, I developed these features to adapt them to the functionality of the BotStudent, this is the list of commands that I thought were useful in that sense:

- **/answer**: this command allows you to answer the questions from the professors, the bot will ask via a ReplyKeyboard which question the professor wants to answer (among the unanswered questions), the answer will be added to the database and all the students who have asked the question will be notified of the answer.

- **/report**: allows you to report a bug to the admin, the message seen by the admin will be of the type "teachers: XXX" with the reported bug in place of the XXX. This message is sent only if a similar one does not already exist in the database.

- **/list**: this command sends a txt file to the user with all topic questions with the answer already recorded.

- **/free_list**: this command sends a txt file to the user with all topic questions still waiting to be answered.
- **/ban**: this command is used to ban a topic question, the bot will ask via a ReplyKeyboard which topic question the professor wants to ban (among the unanswered questions), the students interested in the question will be notified.
- **/ban_list**: this command sends a txt file to the user with with all topic questions banned.
- **/sban**: this command is used to ban a topic question, the bot will ask via a ReplyKeyboard which topic question the professor wants to sban (among the banned questions), the students interested in the question will be notified.
- **/change**: this command allows the teacher to change an answer to a question selected by the teacher. First the bot asks you to select the question (among the answered questions) and then to enter the new answer.
- **/delete**: cancel the initial chat set up, the chat will no longer receive notifications from the bot.
- **/hints**: this command sends the user a file with a list of recommended questions for the topic, the recommended questions are questions taken from the other languages of the topic and translated into the language used by the user.
- **/add_hint**: this command creates a ReplyKeyboard with hints created in the same way as the /hints command. The professor choosing a suggestion adds it to the database.
- **/change_lang**: this command allows the user to change the language, the possible languages will be shown in a ReplyKeyboard and will always be: Deutch, English, French, Italian and Spanish.
- **/change_role**: this command allows the teacher to change roles, if a teacher presses this button he becomes a collaborator and if a collaborator presses this button he becomes a teacher.

11.6 BotCreation

This bot was created to general new bots for students and generate new credentials for professors, this is the command list of this bot:

- **/new_bot**: this command allows you to create new bots for students, first asking the user to enter the topic name (name of at least 5 characters of type [a-zA-Z0-9] or spaces), then subsequently to enter the token created by bot father, then the bot generates passwords that it communicates to the user and the admin via BotPwd. After this process it generates a message with an InlineKeyboard with 2 buttons: one sends to the BotTeacher and one to the BotStudent of the topic.
- **/delete_bot**: this command allows you to delete a BotStudent related to a topic, first the bot asks for the name of the topic, then the password, if the password matches the topic the relative bot is deleted.
- **/change_pwd**: this command allows you to change the password of the topic, first of all the bot makes you choose the topic and then makes you type the password, if it is correct it notifies both the user and the admin with the new password via BotPwd. After this process it generates a message with an InlineKeyboard with 2 buttons: one sends to the BotTeacher and one to the BotStudent of the topic.

N.B. When you have to type a password, the "Forgot password?" button appears on a Reply Keyboard, if you press a notification with the name of the user and the name of the desired topic is sent to the admin.

11.7 The cancel button

There are 2 ways to cancel a command sent: either you send another command or you can click on the Cancel button in the ReplyKeyboard. This button always appears in the intermediate stages of the commands even in those where a ReplyKeyboard would not normally be needed.

11.8 Change language between commands

Sometimes some commands may seem slower than others, this happens because in order not to burden the RAM too much the program loads only one language at a time.

```
1 import spacy
2
3 setted_lang=None
4
5 switch_nlp={
```

```
6       "it":it_fun,
7       "de":de_fun,
8       "fr":fr_fun,
9       "en":en_fun,
10      "es":es_fun
11  }
12
13  def set_nlp(lang):
14      if setted_lang != lang:
15          switch_nlp.get(lang,lambda : None)()
16          setted_lang=lang
```

The bot switches languages with a function similar to the one above (a similar function can be found in the Language class).
So the bot initializes the value of setted_lang to None, every time you need to do an operation that must make use of the language, the set_nlp function is previously called, passing the string with the initials of the language to be set to the function, if the value is the same nothing happens to the one already set, otherwise the new language is set with a slight delay.
The switch_nlp variable is an associative array that takes a string and returns a function, the return functions are the same as those illustrated in chapter 4 for setting languages.

11.9 The files with the questions

The question files are txt files with each line a question or each line is formatted "question -> answer" if you need to write the question in the file as well.

Chapter 12

Import and export Json

In my work I have created 2 additional files that do not interface at all with the rest of the classes. These files are used to upload and download Json files to the database.

12.1 The structure of the Json

This json works on the question part of the database and can have different types of structure, certainly the most complex is the one below:

```
{
    "topic1":
    {
        "de":
        {
            "question1":"answer1",
            "question2":"answer2",
            ...
            "question(k-1)":"answer(k-1)",
            "questionk":"answerk"
        },
        "en":
        {
            ...
        },
        "es":
        {
            ...
        },
        "fr":
        {
```

```
                ...
            },
            "it":
            {
                ...
            }
        },
        "topic2":
        {
            ...
        },
        ...
        "topic(k-1)":
        {
            ...
        },
        "topick":
        {
            ...
        }
}
```

So when you run the export.py program the database is downloaded in the previous format, while when you run the import.py program the file is always uploaded in that format.
This program admits options very similar to linux ones:

- **-lang <string>**: if this option is specified the syntax of the file changes as follows:

```
{
    "topic1":
    {
        "question1":"answer1",
        "question2":"answer2",
        ...
        "question(k-1)":"answer(k-1)",
        "questionk":"answerk"
    },
    "topic2":
    {
        ...
    },
    ...
    "topic(k-1)":
    {
        ...
```

```
    },
    "topick":
    {
        ...
    }
}
```

The language is not specified because only the selected language is updated.

- **-topic <string>**: if this option is specified the syntax of the file changes as follows:

```
{
    "de":
    {
        "question1":"answer1",
        "question2":"answer2",
        ...
        "question(k-1)":"answer(k-1)",
        "questionk":"answerk"
    },
    "en":
    {
        ...
    },
    "es":
    {
        ...
    },
    "fr":
    {
        ...
    },
    "it":
    {
        ...
    }
}
```

The topic is not specified because only the selected topic is updated.

- **-name <string>**: with this command you can specify the name of the file where you take the Json for the upload or where you save the Json for the download, if not specified the file will have the name "questions.txt".

N.B. You can specify more than one option and if both the topic and the language option are specified, the Json structure will be as follows:

```
{
    "question1":"answer1",
    "question2":"answer2",
    ...
    "question(k-1)":"answer(k-1)",
    "questionk":"answerk"
}
```

Bibliography

[1] Gorrino Federico Silvio. *Bot*. URL: https://github.com/FedeIng/tesi (cit. on p. 2).

[2] Gorrino Federico Silvio. *Database*. URL: https://console.firebase.google.com/u/0/project/tesi-database/database/tesi-database/data (cit. on p. 2).

[3] Wikipedia. *Telegram(software)*. URL: https://en.wikipedia.org/wiki/Telegram_(software) (cit. on p. 3).

[4] Wikipedia. *WhatsApp*. URL: https://en.wikipedia.org/wiki/WhatsApp (cit. on p. 5).

[5] Wikipedia. *Slack(software)*. URL: https://en.wikipedia.org/wiki/Slack_(software) (cit. on p. 5).

[6] Wikipedia. *Discord(software)*. URL: https://en.wikipedia.org/wiki/Discord_(software) (cit. on p. 5).

[7] Wikipedia. *WhatsApp*. URL: https://en.wikipedia.org/wiki/WhatsApp (cit. on p. 6).

[8] Telepot. *Telepot*. URL: https://telepot.readthedocs.io/en/latest/ (cit. on pp. 8, 13).

[9] Telegram. *Message*. URL: https://telegram.org/file/811140352/1/wxzxTmvS07c.156395/122fd75657c85aef81 (cit. on p. 9).

[10] Telegram. *ReplyKeyboard*. URL: https://i.stack.imgur.com/HoxWF.jpg (cit. on p. 10).

[11] Telegram. *InlineKeyboard*. URL: https://core.telegram.org/file/811140217/1/NkRCCLeQZVc/17a804837802700ea4 (cit. on p. 10).

[12] telepot. *exception*. URL: https://telepot.readthedocs.io/en/latest/_modules/telepot/exception.html (cit. on p. 15).

[13] Wikipedia. *Semantic similarity*. URL: https://en.wikipedia.org/wiki/Semantic_similarity (cit. on p. 16).

[14] Wikipedia. *spaCy*. URL: https://en.wikipedia.org/wiki/SpaCy (cit. on p. 17).

[15] spaCy. *Word Vectors and Semantic Similarity*. URL: https://spacy.io/usage/vectors-similarity (cit. on p. 17).

[16] ChatCompose. *How to use Chatbots for Education and Learning*. URL: https://www.chatcompose.com/chatbot-learning.html (cit. on pp. 22, 23).

[17] Python. *Datamodel*. URL: https://python.readthedocs.io/en/latest/reference/datamodel.html (cit. on p. 26).

[18] Wikipedia. *Firebase*. URL: https://en.wikipedia.org/wiki/Firebase (cit. on p. 36).

[19] pypi. *firebase 2.2.0*. URL: https://pypi.org/project/firebase/2.2.0/ (cit. on p. 36).