UNIVERSITATEA DE VEST DIN TIMIŞOARA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
PROGRAMUL DE STUDII DE LICENŢĂ: Informatică

LUCRARE DE LICENŢĂ

COORDONATOR:
Conf. Dr. Onchiș Darian

ABSOLVENT:
Babuc Diogen

TIMIŞOARA
2021

UNIVERSITATEA DE VEST DIN TIMIŞOARA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
PROGRAMUL DE STUDII DE LICENŢĂ: Informatică

Chatbot trilingv erudit și inteligent

COORDONATOR:
Conf. Dr. Onchiș Darian

ABSOLVENT:
Babuc Diogen

TIMIŞOARA
2021

Abstract

Through the bachelor's thesis, skills are acquired for the creation and development of a software product, including both the front-end and the back-end parts. This bachelor's thesis is preponderantly practical. The created chatbot tends to be useful to society, as are other chatbots that perform the tasks proposed by the staff of an organization.

The application associated with this work involves the advanced use of the Python programming language, of the libraries available in this language. The application uses methods and classes for machine learning and for text processing in natural language through the interaction between an intelligent system and a person. On the other hand, using the micro-web framework, *Flask*, the web application routes, data registration forms, and database models are built. The *SQLite* database is a relational one. Among other things, the *username* column in the *user* table, which has the constraint to have unique values, is a foreign key in the *message* table, which is intended for the transfer of messages between users on a group communication platform.

The platform is called *intellBot TK* and allows users, through web sockets, to communicate with each other. The platform is intended for erudite users, who are preparing for quizzes or various contests of general knowledge. Web technologies are useful for representing data graphically and for providing features that could not be achieved using the Flask framework.

In this documentation, there are comparisons with other solutions to a similar problem, to the problem of creating and developing a chatbot so that it is intelligent and conversational with high accuracy and a high ability to recognize and give meaning to human words. The built-in chatbot also provides the user, as appropriate, theoretical concepts in the field of *eXplainable artificial intelligence* (XAI), by displaying the definition and URL.

Thus, a chatbot named XAIBOT was built for conversations that accumulate knowledge from multiple domains and reproduces them in conversations with users, at the right times, in this way becoming erudite. He is also intelligent. He should be able to discuss with the user about claims, theorems, definitions of terms, or various events.

The application is one that runs both online and offline, due to the progressive web application (PWA) element called *Background Sync*. The application may also be available on mobile devices due to the responsive property.

Rezumat

Prin lucrarea de licență se dobândesc competențe pentru crearea și dezvoltarea unui produs software incluzând și partea de front-end și cea de back-end. Această lucrare de licență este una preponderent practică. Chatbot-ul creat tinde să fie și el util societății, cum sunt și alți chatboți care îndeplinesc sarcinile propuse de personalul unei organizații.

Aplicația asociată acestei lucrări implică utilizarea avansată a limbajului de programare Python, a bibliotecilor de care dispune acest limbaj. În aplicație se folosesc metode și clase pentru învățarea automată și pentru procesarea textului în limbaj natural prin interacțiunea dintre un sistem inteligent și om. Pe de altă parte, folosind cadrul de lucru micro-web, *Flask*, se construiesc rutele aplicației web, formele de înregistrare a datelor și modelele pentru baza de date. Baza de date *SQLite* este una relațională. Printre altele, coloana *username* din tabelul *user*, care dispune de constrângerea de a avea valori unice, este cheie străină în cadrul tabelului *message*, care este destinat transferului de mesaje între utilizatori pe o platformă de comunicare în grup.

Platforma este denumită *intellBot TK* și permite utilizatorilor, prin intermediul prizelor web (Web Sockets), să comunice reciproc. Platforma este destinată utilizatorilor eruditi, care se pregătesc pentru quiz-uri sau diferite concursuri de cunoștințe generale. Tehnologiile web sunt utile pentru a reprezenta grafic datele pe bordul de lucru și pentru a oferi și unele funcționalități care nu au putut fi atinse utilizând cadrul Flask.

În această documentație există comparații cu alte soluții la problema similară, la problema de creare și dezvoltare a unui chatbot astfel încât acesta să fie inteligent și conversațional cu o acuratețe înaltă și cu o capabilitate înaltă de a recunoaște și a da sens cuvintelor transmise din partea omului. De asemenea, chatbot-ul construit furnizează utilizatorului, după caz, concepte teoretice din domeniul inteligenței artificiale explicabile (XAI), prin afișarea definiției și a adresei *URL*.

Astfel, s-a construit un chatbot pentru conversații care acumulează cunoștințe din domenii multiple și le reproduce în conversațiile cu utilizatorii, la momentele potrivite, devenind astfel erudit. De asemenea, el este și inteligent. Ar trebui să fie capabil să poată discuta cu utilizatorul despre revendicări, teoreme, definiții ale termenilor sau despre diferite evenimente.

Aplicația este una care rulează și online și off-line, din cauza elementului de aplicație web progresivă denumit *Background Sync*. Aplicația poate fi disponibilă și pe dispozitivele mobile din cauza proprietății de *responsive*.

Cuprins

Introducere

Obiectivul principal al lucrării de licență pentru tema *Chatbot trilingv erudit și inteligent* îl reprezintă utilizarea de software, dezvoltarea de software, implementarea, testarea și învățarea.

Scopul acestei lucrări de licență este îmbunătățirea funcționalității a unui chatbot denumit XAIBOT, totodată și asistent virtual care furnizează date despre unele concepte XAI, utilizând algoritmi și modele din domeniul învățării automate și din cadrul de procesare al textului în limbaj natural pentru a simula un comportament uman conversațional. Pentru ca aplicația să fie disponibilă și accesibilă tuturor, s-a ajuns la concluzia că ar fi necesar să existe un site web și o aplicație pentru dispozitivele *Android*, pentru început. Iar după ce aplicația a fost transformată într-o aplicație web care rulează pe serverul local, o parte din aceasta - strict partea cu chatbot-ul trilingv, însă expert în conceptele din domeniul XAI - a fost lansată pe Heroku și pe platforma de competiții în domeniul Inteligenței Artificiale, denumită *XAION*. Adresa URL a aplicației web a fost introdusă ca și parametru pentru aplicația Android, pe platforma *AppsGeyser* de unde s-a descărcat aplicația Android aferentă site-ului web. Principalul scop la crearea și dezvoltarea chatbot-ului era ca asistentul virtual, XAIBOT, să devină conversațional pentru discuțiile din domeniul inteligenței artificiale explicabile.

Totuși, cu timpul s-a ajuns la ideea că chatbot-ul ar putea fi dezvoltat astfel încât să fie trilingv, să discute în limba română, limba sârbă și limba engleză deoarece și autorul cunoaște și utilizează aceste trei limbi. Treptat, baza de cunoștințe construită prin stocarea datelor în fișierele *Yaml* s-a extins, iar chatbot-ului i-au fost acordate cunoștințe din geografie, astronomie, istorie, sport, cinematografie și altele. Chatbot-ul va trebui să învețe din aceste date, având grijă la diferitele categorii de date, pentru a putea furniza un răspuns clar la cele transmise din partea utilizatorului.

De menționat este că la începutul dezvoltării aplicației, atunci când aceasta a fost vizibilă doar în consolă, s-a utilizat denumirea *REFBOT* deoarece chatbot-ul detecta doar referințele pentru un text trimis de utilizator, iar ulterior numele asistentului virtual s-a modificat în *XAIBOT* datorită faptului că acesta devine erudit și inteligent, iar principala sa sferă de expunere este chiar inteligența artificială explicabilă (XAI).

Capitolul 1

Descrierea problematicii

În acest capitol se va descrie problematica aplicației într-un mod general și concret. Se vor indica posibilele moduri de soluționare ale problemei și cazul de utilizare din aspectul programatorului, care încearcă să facă o descriere a secțiunilor din aplicație care trebuie conspectate pe baza a mai multor surse.

1.1 Descrierea generală prin fluxuri de informații

În aplicație, s-a pornit de la un chatbot realizat la Stagiul de Practică care a acumulat cunoștințe doar din inteligența artificială și le-a exprimat în conversațiile cu utilizatorul.

Una dintre primele întrebări care trebuiau să apară la această secțiune este: "Unde ar trebui și cum să fie stocate și structurate datele de învățare ale chatbot-ului?". Ideea inițială era să se creeze un adaptor de stocare pentru stocarea datelor, în care chatbot-ul să învețe lucruri noi, iar baza sa de cunoștințe să se expandeze. La partea de expandare a bazei de date, prin intermediul obiectului de tip ChatBot din librăria *chatterbot*, au apărut primele situații grave. Baza de date se supraîncărca într-un ritm foarte rapid astfel încât nu era prea posibilă urcarea aplicației pe o platformă. Site-ul mergea tot mai încet. Pentru a elimina aceste evenimente nefavorabile noua bază de cunoștințe au devenit fișierele Yaml, fiecare fișier menit pentru stocarea de informații într-o singură limbă. Principiul care a rămas până la final este cel cu fișierele *Yaml* - pentru care s-a importat librăria *PyYaml* în Python pentru a explora și a manipula datele. Cele trei fișiere Yaml i-ar permite asistentului virtual să acumuleze cunoștințe generale și din alte domenii în afară de ramurile inteligenței artificiale și informatică. De asemenea, chatbot-ul poate detecta și furniza referința pentru un termen din învățarea automată, cum ar fi, de pildă, modelul LIME. La început s-au folosit doar câteva referințe. În cazul în care propoziția detectată nu are referință, adică nu este găsită prin utilizarea algoritmului de la funcția `cosine_similarity()` a librăriei *Scikit-learn* [BKL19], se va afișa doar propoziția detectată.

Încă o întrebare care s-ar putea pune este cum să se construiască chatbot-ul astfel încât să vorbească trei limbi distincte în paralel. Acest lucru s-ar putea realiza dacă s-ar combina cele trei fișiere Yaml într-unul singur. Totuși, utilizând acest procedeu, ar putea apărea o problemă de suprapunere a datelor deoarece chatbot-ul nu este deloc instruit cu datele care nu există în mulțimea de date.

Chatbot-ul este unul conversațional, însă și tranzacțional deoarece comunică cu un

sistem extern, de unde și preia informații și le furnizează către utilizator.

Cum să se construiască un chatbot conversațional sau cum să fie reprezentat fluxul conversației, erau câteva întrebări care trebuiau conspectate. Bordul principal de lucru, în care să fie vizibilă funcționalitatea chatbot-ului, poate reprezentat precum o cutie conversațională, poate fi similar cu aplicația de WhatsApp, Viber sau Messenger Chats deoarece este vorba de o pagină în care utilizatorul discută cu un automat umanizat. În ceea ce privește fluxul de informații, la bunul început sistemul inteligent face cunoștință cu utilizatorul. După ce utilizatorul și-a introdus numele, chatbot-ul îl întreabă cu ce se ocupă, toate cu scopul de a simula comportamentul uman la chatbot și pentru a relaxa utilizatorul. Totuși, chatbot-ul comportându-se precum o persoană grăbită și deseori direcționată către subiectul principal de elaborare, îi propune utilizatorului un flux în care numiții discută precum niște eruditi sau sunt concentrați pe modelele și algoritmii de învățare automată. Ceea ce este important este că botul tinde spre dezvoltarea unei personalități proprii. El pune întrebări, dar și răspunde la ele. Încă o întâmplare delicată care trebuia luată în considerare cu atenție este modul în care chatbot-ul este poziționat spre partea de referințe (link-uri). Inițial au existat două căii pe care le putea considera utilizatorul: cea de dezvoltare a cunoștințelor universale și cea de furnizare a link-urilor. Pentru a nu intra în complicații și pentru a nu apărea erori, evenimentul pomenit se utilizează împreună cu fluxul conversațional erudit.

Când este util să existe sectorul tranzacțional de date? Cum ar trebui gestionat acesta? Întrebările, sau nelămuririle următoare erau analizate din punctul de vedere al programatorului. Sectorul tranzacțional este folositor pentru a expanda cu mult cunoștințele sistemului virtual automat. Atunci acest sistem poate avea o aplicabilitate mai concretă, prin simpla conexiune dintre funcționalitatea sistemului inteligent și o altă interfață a aplicației programabile (API) deja existentă. Câteva exemple ar fi știrile, calendarul, *Gmail*, sau cumpărăturile online. Acest procedeu rămâne de conspectat mai în detaliu la studiile de master. La aplicația aferentă lucrării de licență s-au introdus în baza din Yaml, la categoriile corespunzătoare, link-urile care informează mai mult despre un model sau algoritm al sistemelor inteligente, cum ar fi, de pildă, *K cei mai apropiați vecini* sau *rețeaua neuronală sub forma unui graf.*

Pentru ca chatbot-ul creat să fie util și aplicat pe o platformă concretă, s-a implementat și o platformă de mesagerie online și off-line pentru utilizatorii despre care se presupune că au dobândit pe parcursul vieții lor multe cunoștințe universale demne de laudă. Platforma permite mai multor utilizatori să comunice simultan. Aceștia primesc notificări atunci când un utilizator trimite un fișier sau un mesaj în chat. Utilizatorii vor avea posibilitatea să-și pună diferite întrebări, sau chiar să organizeze mici evenimente în care să-și testeze cunoștințele actuale sau chiar să și le extindă. Evident, pentru a distinge utilizatorii, ei vor avea posibilitatea ca la accesarea aplicației să se înregistreze dacă deja nu au un cont. În cazul în care au un cont creat, pot accesa aplicația. La partea de înregistrare s-au introdus validări pentru adresa de e-mail, pentru lungimea parolei, unicitatea numelui de utilizator pe când la partea de logare se încearcă potrivirea datelor cu cele deja stocate în baza de date. Pe pagina de profil al utilizatorului apar cele trei opțiuni pentru selecția limbii în care se va discuta cu chatbot-ul.

Dacă în chat apare vreo întrebare care îi este nefamiliară utilizatorului sau la care nu știe răspunsul, poate să îi adreseze acea întrebare chatbot-ului chiar dacă nu are acces la *Wi-Fi*. Când utilizatorul intră pe pagina web unde tinde să discută cu XAIBOT, îi apare data și timpul când a început conversația. După ce utilizatorul trimite un

mesaj, iarăși îi apare data și timpul când l-a trimis. De partea stângă a mesajului se găsește o imagine cu asistentul virtual, respectiv o imagine a utilizatorului. Sunt indicate și momentele când chatbot-ul tastează un mesaj, pentru a interpreta discuția reală, om la om.

1.2 Ilustrare prin cazuri de utilizare

În această secțiune se va vizualiza modul de abordare al problemei preluat de programator. Anume, programatorul mai întâi s-a informat despre diversele biblioteci care sunt puse la dispoziție tuturor oamenilor de știință ai informaticii. Este vorba de cărțile *Numerical Python* scrisă de Robert Johansson și *Natural Language Processing with Python* scrisă de Bird, Klein și Loper. Informațiile cuvioase mai departe vin din articolul *Personal Assistant* construit de Nair, Sathya și Johnson și articolul amplu publicat în anul 2020, numit *Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI.*

Modul de abordare al problemei presupune și dezvoltarea de software, care include procesul de extragere sau construire al datelor și partea de proiectare a interfeței grafice a utilizatorului. Extragerea datelor se efectuează prin intermediul bibliotecilor și funcțiilor din Python, iar transferul de variabile unde sunt stocate rezultatele extragerii de date se efectuează prin intermediul tehnologiei Flask, mai precis utilizând *render_template()*, iar utilizarea efectivă a variabilelor în tehnologiile web - în care, după manipularea acestei variabile, se pregătesc datele pentru reprezentarea grafică - se face prin comenzile oferite de motorul extensibil de modelare, $Jinja2$.

Ulterior, programatorul a trebuit să se gândească la un flux al aplicației, adică la modul și ordinea de accesare a paginilor în aplicație. Astfel, la accesarea site-ului, utilizatorii mai întâi se înregistrează sau loghează, iar apoi vor fi transferați pe pagina de profil, unde vor avea șansa să acceseze fie chatbot-ul în limba dorită, fie platforma de mesagerie de tip difuzare.

Ultimul parte a creării site-ului presupune transferul acestuia pe Heroku, utilizând diferite librării, precum și comenzi aferente sistemului de surse deschise, *git*, în terminal, pentru ca site-ul să fie accesibil pentru toți.

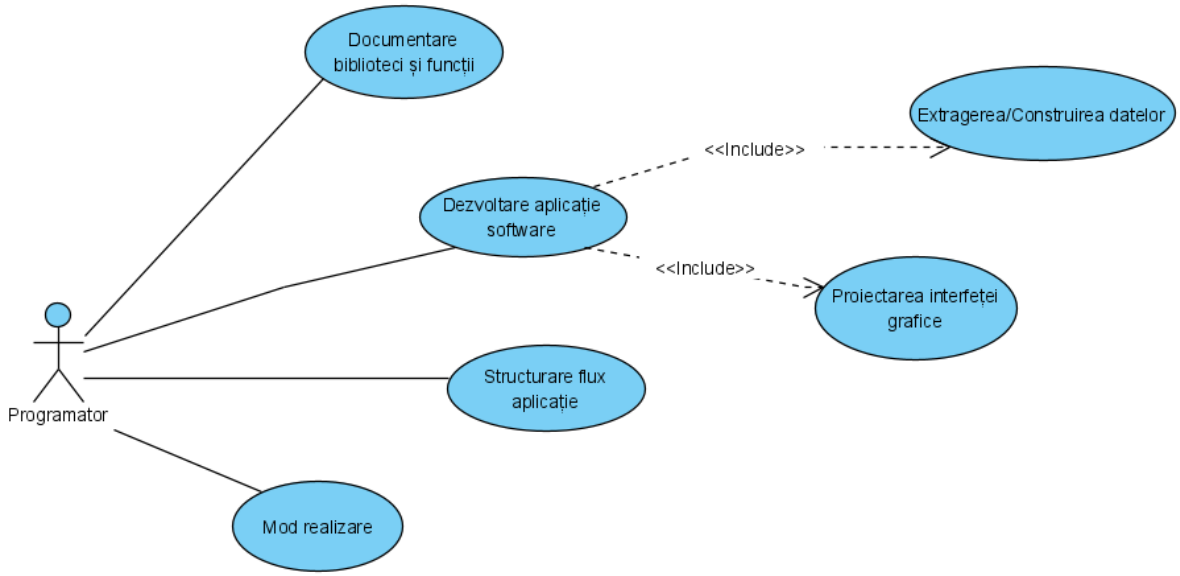

Figura 1.1: Cazul de utilizare destinat programatorului

Capitolul 2

Soluții și abordări similare

În acest capitol se va face o analiză amplă asupra celor studiate, dezvoltate, implementate și efectuate de savanții din informatică sau alți oameni care s-au dedicat sau se dedică și în continuare învățării automate (profunde), procesării textului în limbaj natural și modalităților de reprezentare a datelor de ieșire, precum și structurarea interfeței grafice a utilizatorului. Se va face o comparație dintre diferitele abordări pentru a vedea ceea ce distinge chatboții implementați asemănător din punctul de vedere al utilizării și al performanței.

2.1 Descrierea aplicațiilor și a abordărilor similare

Sathya, Johnson și Nair au descris în articolul *Chatbot as a Personal Assistant* că boții meniți pentru discuții sunt programați astfel încât să creadă că discută cu oameni[NJG18]. Principalul avantaj al caracteristicilor oferite de programator pentru un chatbot este acela că i se poate construi o personalitate virtuală proprie, specializată într-un singur domeniu, sau în mai multe domenii. Articolul lor propune o idee despre un chatbot ca asistent personal, utilizând metode și metodologii din sfera inteligenței artificiale. Un astfel de bot poate ajuta utilizatorul în diferite cazuri și la diferite momente[ADRS+20]. Chatbot-ul pe care ei l-au construit este *ResumeBot*. Deci, este vorba de o autobiografie profesională sub forma unui chatbot. Acest tip de chatboți sunt utili atunci când individul se candidează pentru un loc de muncă la o firmă cutare.

ResumeBot-ul lor va ajuta recrutorul să recunoască personalitatea candidatului, la fel și calificările acestuia și datele personale. Chatbot-ul va comunica cu utilizatorul prin intermediul limbajului *nltk*[BKL19] și al fișierului $AIML$. Ei definesc chatbot-ul drept un program computațional care ajută indivizii să comunice cu alții prin mesaje. Într-un mod teoretic, dar și practic în lumea virtuală, imita comportamentul uman. Reproduc modul de acționare uman într-o conversație cu un alt om[NJG18]. Autorii textului menționează că este straniu faptul că aplicațiile chatbot sunt considerate noi browser-e și site-uri web, însă acesta este adevărul pur, în conformitate cu comportamentul lor virtual. Exemple universal cunoscute de boți de comunicare sunt Cortana, Siri, Alexa. Și pe Messenger-ul de la Facebook utilizatorii pot construi un chatbot fie conversațional, fie tranzacțional, care să le permite, de pildă, să construiască o statistică. Prin utilizarea tehnologiei de care dispune mediul inteligenței artificiale, se poate scoate în evidență și propriul nostru profil pentru munca profesională sau doar

hobby. Inteligență artificială permite îmbunătățirea profilului[KZH17]. Bine de știut este că în lume există tot mai mulți boți de comunicare care oferă un suport firmelor sau organizațiilor de business pentru a le ajuta în relația lor cu clienții[NJG18].

Pentru a structura un model de atenție, utilizând rețeaua neuronală recurentă, cu rolul de a corecta problemele de memorie care apar la un sistem inteligent, autorul articolului *Deep Learning for NLP: Creating a Chatbot with Keras*, Jaime Zornoza, s-a gândit să folosească librăria *Keras* în limbajul de programare Python[Zor19]. Primul procedeu pe care l-a abordat a fost crearea unui model cu straturi secvențiale. După aceasta, a compilat întreaga structură a rețelei pentru a transforma straturile într-o matrice de operații. Următorul algoritm care trebuie aplicat, spuse Zornoza, este cel de antrenare și potrivire a datelor[Zor19] din rețeaua edificată. Chatbot-ul dezvoltat va simula și el un comportament uman, urmat de o personalitate de om curios. Creierul artificial al sistemului este construit din neuronii legați într-un graf dirijat. În aplicație, *Keras* fiind doar o interfață aplicativă de programare, se folosesc cadrele de lucru *Tensorflow*, *Theano* și *CNTK* (Cognitive Toolkit), pe când, pentru manipularea textului, se utilizează librăria *NLTK*, cea de procesare a datelor în limbaj natural.

Despre biblioteci s-a studiat din cartea lui Robert Johannson, numită *Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib*. De aici s-a ajuns la concluzia că se poate folosi librăria *Scikit-learn* pentru învățarea automată. Totuși, mai există și două librării ($Pandas$ și $Matplotlib$), menționează Johansson, care pot fi utile, prin proiectarea și afișarea rezultatelor, în dezvoltarea unui produs soft.

Librăria *Pandas* este un cadru pentru analiza și procesarea datelor în *Python*. Pentru a importa librăria, se utilizează comanda *import pandas*, iar aliasul care se acordă convențional acestei librării este `pd`. Această librărie oferă structuri de date și metode pentru reprezentarea și manipularea datelor. Două cele mai importante structuri de date în librărie sunt obiectele *Series* și *DataFrame*. Obiectele Series sunt utilizate pentru a reprezenta datele sub forma unei serii, iar DataFrame pentru reprezentarea sub forma unui tabel[Joh19].

Matplotlib este o librărie pentru crearea vizualizărilor statice, animate și interactive. Librăria produce figuri de o mare calitate și dispune de pachetul *pyplot*, care se importează prin comanda: *import matplotlib.pyplot*, iar aliasul convențional este `plt`. Folosind modulul *pyplot* din librăria *Matplotlib*, se pot transpune datele în grafice pentru a obține o figură. Fiecare figură conține unul sau mai multe sisteme de coordonate. Pentru a crea un grafic, se folosesc următoarele linii de cod: $fig, ax = plt.subplots()$, unde se indică crearea unui singur sistem de coordonate $ax.plot([1, 9, 2, 3], [3, 7, 5, 2])$, sau direct linia de cod: $plt.plot([1, 9, 2, 3], [3, 7, 5, 2])$. Pentru a afișa o figură fără sistemul de coordonate, se utilizează doar comanda $plt.figure()$. Este cunoscut că un sistem de coordonate conține două sau trei axei (depinde dacă e sistem 2D sau 3D). Din această cauză se setează limitele datelor introduse în sistem prin metodele *ax.Axes.set_xlim()*, *ax.Axes.set_ylim()* și *ax.Axes.set_zlim()*, unde `ax` este o figură cu un singur sistem (un obiect din modulul *matplotlib.pyplot*). Pentru fiecare sistem de coordonate se poate seta titlul prin metoda *set_title()*, axa Ox prin metoda *set_xlabel()*, axa Oy prin metoda *set_ylabel()* și, dacă este necesar, axa Oz cu *set_zlabel()*. Cea mai simplă metodă pentru a crea o grilă, de pildă $2x2$ pentru sistemul de coordonate, este aceea care implementează comanda: *fig, axs = plt.subplots(2, 2)*. Pe lângă crearea interfeței cu *pyplot*, librăria matplotlib deține posibilitatea de a crea interfața orientată obiect. La interfața orientată obiect există și linia de cod în care se creează figura

și sistemul/sistemele de coordonate. Dacă există mai multe grafice într-un singur sistem, atunci se folosește o legendă pentru a deosebi dreptele sau curbele. Numele la grafice se acordă cu ajutorul (prin inițializarea lui) atributului *label*, iar metoda pentru afișarea căsuței cu numele și forma (sau culoarea) graficelor este *plt.legend()*. Mai există o metodă, numită *linspace()*, din librăria *numpy*, care specifică intervalul valorilor pentru una (sau mai multe) dintre axe și numărul de zecimale[Joh19].

Totuși, într-o lume în care site-urile web sunt foarte utilizate și ușor de accesat, nu există o relevanță prea mare de a reprezenta datele doar pe interfața grafică a unui limbaj de programare utilizând *toolkit*-uri GUI, când vine vorba de un proiect mai amplu cum este lucrarea de licență.

Pe de altă parte, o librărie deseori utilizată în aplicație pentru crearea și dezvoltarea chatbot-ului ca și asistent virtual trilingv erudit și inteligent, este librăria $Scikit-learn$. $Scikit-learn$ conține o mulțime de algoritmi pentru $machine-learning$, incluzând regresia, clasificarea și reducerea dimensiunii. Pentru a importa biblioteca, este necesar să tastăm comanda import sklearn, iar dacă dorim să se importă vreun modul (de ex. *datasets, metrics, tree, svm, cluster*, etc) din această bibliotecă, se utilizează comanda: *from sklearn import modul*. Modulul *model_selection* din *sklearn* conține funcții care lucrează cu validarea bazată pe încrucișare. La etapa de preprocesare a problemei de învățare automată, trebuie abordată extragerea cerințelor. Trebuie creată cerința corespunzătoare a matricelor care se transmit spre unul sau mai mulți algoritmi ai învățării automate, care sunt codați în librăria Scikit-learn. Modulul din *sklearn* numit *feature_extraction* servește la aplicațiile de învățare automată precum librăria *Patsy* în modelarea statistică[Joh19], mai ales atunci când apar probleme legate de text sau imagini la învățarea automată. Folosind $feature\_extraction$, se poate asambla automat proiectarea matricelor din diverse surse cu date. Regresia cuprinde partea centrală a învățării automate și a statistici de modelare. Regresia este importantă deoarece ea poate prezice foarte bine noile observații. De pildă, dacă există un set mare de cerințe și un număr scăzut de observații, se poate aplica regresia pentru a o potrivi cu datele, într-un mod ideal, fără a mai fi nevoie de prezicerea noilor valori. Pentru a aplica tehnica de regresie, este necesar să se importe modulul *datasets* din `sklearn` și să se acceseze metoda *make_regression*(), unde se indică numărul de intrări, numărul de cerințe și numărul de informative. Datele de ieșire vor fi doi vectori, X_all și y_all cu dimensiunile specificate la inițializarea atributelor din metoda *make_regression()*. Și tehnica de clasificare cuprinde partea centrală a învățării automate. Clasificarea se utilizează la modelul regresiei logistice pentru a ordona pe categorii observațiile[Joh19]. Important de menționat, regresia logistică este folosită pentru aceeași sarcină de clasificare la învățarea automată, având mai mulți algoritmi distincți pentru clasificare. De pildă arborii de decizie, metodele pădurii aleatoare, metodele celui mai apropiat vecin, SVM și altele. Fiecare problemă de clasificare poate fi rezolvată prin utilizarea diferitelor metode oferite de API-ul unificat din sklearn, într-un mod interschimbabil. Pentru a încărca setul de date, se apelează metoda *load_iris()* din modulul *datasets* din librăria *sklearn*. Tipul de date returnat va fi un obiect container (mai numit și obiect *Bunch*) care conține și date și metadate, sau mai precis *sklearn.datasets.base.Bunch*. Când vine vorba de regresia logistică, menționează Johansson, pentru utilizare se folosește clasa *LogisticRegression* din modulul *linear_model*. Atributele predefinite pe care le utilizează clasa sunt $fit\_intercept$, $intercept\_scaling$, $penalty$, $solver$, $random\_state$, $verbose$, max_iter, $multi\_class$, dual și altele, care sunt inițializate la constructor[Joh19]. Pen-

tru a utiliza o modalitate de antrenament virtual, se folosește metoda fit din modulul $linear\_model$. Metoda fit are două atribute: X_train și y_train. După ce a început instruirea, există posibilitatea ca utilizatorul să descopere noi observații prin metodele de prezicere. De asemenea, prin intermediul funcției $predict(X\_test)$ se pot compara observațiile obținute la preziceri cu valorile curente obținute din datele de intrare pasate. Reducerea dimensiunii și selecția cerinței sunt două funcționalități importante în aplicațiile cu învățare automată. Modulele care se folosesc sunt $sklearn.decomposition$ și $sklearn.feature\_selection$, care conțin funcții pentru reducerea dimensiunii a unei probleme ce apare în sfera de învățare automată. Un exemplu de principiu ar fi, de pildă, *Principal Component Analysis*, care efectuează o decompoziție cu o singură valoare dintr-o matrice și păstrează doar acele valori care corespund la cei mai benefici vectori.

2.2 Comparația dezvoltării a doi chatboți similari

În această secțiune se va face o analiza paralelă a doi chatboți similari - *ResumeBot* și chatbot-ul implementat cu librăria pentru modele de rețele neuronale, *Keras* - dezvoltați în limbajul de programare Java, respectiv Python.

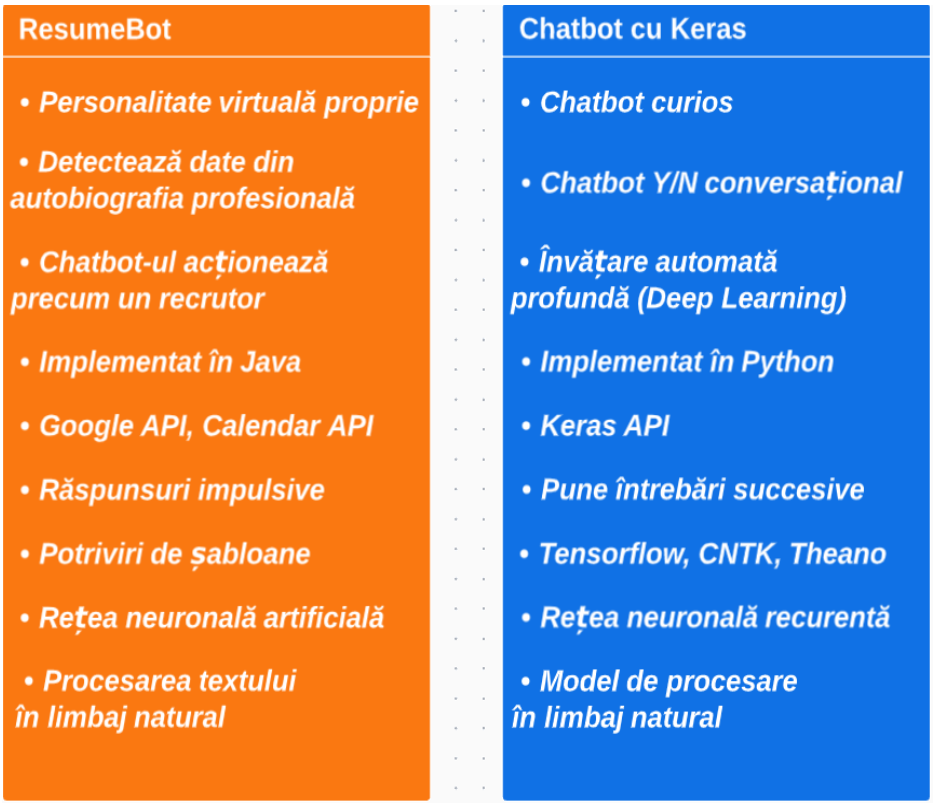

Figura 2.1: Abordări în construirea chatbot-ului

Din figura 2.1, se poate deduce că chatbot-ul implementat cu Keras este unul care folosește structuri pentru rețele neuronale, cu rostul de a extrage date dintr-un text. Această abordare la dezvoltarea chatbot-ului o propune Jaime Zornoza. Chatbot-ul adresează întrebări conectate între ele, deplasându-se de la ramură la ramură pe un

arbore de decizie provizoriu, astfel încât să ajungă la o soluție. La întrebări se poate răspunde doar cu *da* și *nu*. Sistemul inteligent implementează modele de procesare a textului în limbaj natural, la fel ca și chatbot-ul personal dezvoltat de Nair, Johnson și Sathya. Întrebările adresate utilizatorului ar trebui să fie relativ simple, pentru ca acesta să fie capabil să răspundă la ele. Spre deosebire de chatbot-ul recrutor - unde s-au construit în limbajul de programare Java potriviri de șabloane, într-un mod manual - chatbot-ul care dispune de Keras API folosește diferite cadre de lucru pentru învățarea automată profundă[Zor19], cum ar fi *Theano*, *CNTK (Cognitive Toolkit)* sau *Tensorflow*. Pentru acest sistem inteligent s-a creat o rețea neuronală recurentă, care este mai dezvoltată decât cea artificială (simulează în mod virtual informațiile de procesare din creierul omului) folosită la *ResumeBot*. La rețeaua neuronală recurentă vârfurile sunt legate între ele astfel încât să construiască un graf dirijat, fapt care permite ca sistemul inteligent să ajungă la un răspuns și atunci când îi este furnizat un text cu scris de mână sau un conținut audio.

Capitolul 3

Descrierea generală

În capitolul acesta se vor specifica și indica elementele și proprietățile introduse sau folosite de autor pentru a ajunge la soluții pentru problematica specificată în primul capitol. Capitolul al treilea este unul esențial datorită faptului că aici se indică ideile originale ale autorului, precum și modalitățile de soluționare în conformitate cu sectoarele care există în cazul de utilizare din primul capitol al lucrării. Astfel, cazurile de utilizare vor fi extinse și detaliate folosind diagrame de secvență.

3.1 Contribuții personale în scopul soluționării problemei

După cum se poate observa din titlul acestei secțiuni, aici se va săvârși o descriere mai amplă a contribuțiilor autorului asupra reprezentării soluțiilor.

Pentru documentare s-a folosit articolul *Explainable Artificial Intelligence: Concepts, taxonomies, opportunities and challenges toward responsible AI*, cartea *Natural Language Toolkit*, cartea lui Robert Johannson cu numele *Numerical Python*, apoi articolul *Chatbot as a Personal Assistant*. S-a construit un chatbot de bază, care a detectat propoziția în care apare cuvântul introdus ca și dată de intrare, utilizând algoritmul de la funcția `cosine_similarity()`. Pentru acea propoziție s-a verificat referința (dacă există) și s-au indicat date despre referința respectivă. Chatbot-ul a fost dezvoltat astfel încât să fie conversațional, cu comportament uman. Se indică momentele de timp când chatbot-ul tastează un mesaj. Poate termina de scris un mesaj relativ lung în câteva secunde, fapt care demonstrează că este superior oamenilor chiar dacă tinde spre o personalitate pentru a se personifica. Acuratețea chatbot-ului la datele de intrare ale utilizatorului a fost una mediocră. Totuși, chiar dacă textul utilizatorului depășește cunoștințele chatbot-ului, acesta se descurcă atunci când furnizează un răspuns, având un simț al umorului specific, unic.

Deoarece aplicația a rulat doar în consolă, s-a decis să se transfere codul într-un cadru de lucru, cum este Flask, astfel încât să fie posibilă conexiunea dintre partea în care au fost introduse, extrase și procesate datele, și partea în care acestea sunt afișate pe bordul de lucru al unei pagini web care rulează, pentru început, local.

De asemenea, pentru transferul de variabile din partea de back-end în cea de front-end s-a utilizat tehnologia *Jinja*2 și librăria din JavaScript, numită *JQuery*. Atunci când se utilizează JQuery, parametrului funcției *get* îi este asociată ruta construită în Flask. Procesul care s-a săvârșit pentru a transmite codul în Flask presupune instala-

rea librăriei *flask* și a altor librării necesare pentru program și crearea variabilei de tipul clasei Flask, cu parametrul `__name__`. La citirea din fișiere s-a utilizat calea absolută spre fișier, ceea ce a necesitat importul librăriei *os*. Etapa de salutare cu chatbot-ul a fost introdusă direct în funcția `response(user_response)`, deoarece funcția cu numele *get_bot_response()* returnează funcția *response*. Pentru această funcție s-a creat ruta */get*. În funcție se preia textul introdus în câmpul de text pentru transmiterea mesajelor. În fișierul *manipulareYaml.py*, crearea și utilizarea funcției *findvrc()* îi permite lui XAIBOT să găsească propoziția și referința aferentă cuvântului introdus la input, dacă chatbot-ul a detectat în baza sa de cunoștințe (pentru textul utilizatorului) o adresa *http*. Fișierele HTML din directoriul cu șabloane, *templates*, îi permite utilizatorului să observe interfața grafică. Stilul este modelat în fișierele *style.css* din directoriul *static*.

În ceea ce privește baza de date, la rularea aplicației se creează automat baza de date *SQLite*, declarată și configurată utilizând clasa *Config*, creată manual, unde se setează pe valoarea *Adevărat* parametrul de interzicere a cererii de falsificare (CSRF), se indică o cheie secretă de acces la baza de date și se specifică identificatorul uniform al resurselor (URI) pentru baza de date creată prin intermediul instrumentului *SQLAlchemy* care tratează clasele definite manual drept tabele dacă aceste clase implementează clasa *Model* accesată din instanța clasei *SQLAlchemy* din librăria *flask_sqlalchemy*.

Textul scris de utilizator pe interfața grafică poate fi trimis în chat prin apăsarea butonului *Send/Trimite/Pošalji*. De asemenea, încă o idee originală a fost să se creeze secțiunea de cunoaștere dintre sistemul expert și utilizator. Pentru această discuție s-a folosit un nou fișier și un set de metode. Se va parcurge lista cu propozițiile, în general simple, care sunt destinate acestei etape. Metodele esențiale sunt `name()`, `discut()` și `final()`. În *name*() se descoperă numele utilizatorului.

Pentru a porni aplicația, se scrie în *main*() comanda *app.run*(), la care se poate seta și parametrul de depanare și cel de gazdă.

Urcarea aplicației pe un site public s-a efectuat utilizând mediul de dezvoltare integrat, *Heroku*. Ulterior, s-a creat aplicația pentru dispozitive mobile utilizând $WebView$ dintr-un compilator online. Aplicația mobilă, care se actualizează de fiecare dată când se actualizează site-ul, se numește „XAIBOT". S-a obținut o platformă pentru accesarea aplicației și o aplicație mobilă, *Android*.

Pentru secțiunea conversațională, s-a utilizat clasa ChatBot din biblioteca chatterbot; mai precis, s-a declarat un obiect de tipul ChatBot, care are ca și atribute numele și un adaptor de stocare, SQLStorageAdapter din biblioteca storage din chatterbot, care i-a permis botului să stocheze date pentru conversație în SQLite3. Adaptorul de stocare îi oferă instanței clasei ChatBot o interfață pentru a se putea conecta cu diverse backend-uri de stocare. Acest obiect s-a utilizat în Flask la ruta */get*, adică metoda *get_bot_response()*, unde se păstrează într-o variabilă textul introdus de utilizator prin comanda *request.args.get('msg')*, iar pentru a porni conversația în limba engleză discutând despre diferite domenii, se accesează metoda *get_response()* de instanța clasei ChatBot. Textul accesat din interfață este introdus drept parametru.

Totuși, această abordare a fost costisitoare din aspectul timpului de execuție și al memoriei. Din cauza supraîncărcării adaptorului de stocare, aplicația urcată pe site (Heroku) rula tot mai încet. Abordarea a fost îmbunătățită cu biblioteca de învățare automată, numită *Scikit-learn*. Importând modulul bibliotecii *feature_extraction.text*, se folosește clasa *Tfidfvectorizer*. Această clasă permite divizarea directă a tuturor

întrebărilor posibile, din baza de cunoștințe, într-o listă de cuvinte sortate crescător după codul ASCII, prin funcția *get_feature_names()* la care este indicat, ca și parametru, corpusul de date. Pe de altă parte, utilizând metoda *fit_transform()*, sistemul învață vocabularul din baza de cunoștințe sincronizată în trei fișiere *Yaml*.

În cazul în care textul introdus de utilizator este „exit", atunci se iese din flux. O altă categorie, cea cu furnizarea de link-uri și definiții ale modelelor din învățarea automată, dintr-un text, s-a importat biblioteca os pentru a găsi calea directoarelor, iar în directoare și calea absolută spre fișierele dorite, cu țelul de a citi din aceste fișiere. După citire, s-a făcut o conversie din lista cu un singur text amplu, în lista cu propoziții, prin *sent_tokenize(linie)* din biblioteca importată, `nltk`. De asemenea, pentru a grupa diferite forme de cuvinte astfel încât ele să poată fi ușor de analizat, se utilizează clasa *WordNetLemmatizer* din pachetul *nltk.stem*. Funcția principală este response(raspunsul_utilizatorului) definită de programator, unde se găsește și funcția pentru salutarea chatbot-ului cu utilizatorul. Procesul aplicat pentru detectarea propoziției în care se găsește cuvântul/sintagma introdusă este *cosine_similarity()* din modulul *metrics.pairwise* din *sklearn*. Funcția se numește *cosine similarity* deoarece normalizarea euclidiană proiectează vectorii pe sfera unitară, iar produsul lor scalar este cosinusul unghiului dintre punctele notate de vectori. Deci, chatbot-ul va returna propoziția și un link asociat termenului din inteligența artificială, dacă acesta există. Verificarea de existență al protocolului *http* se face în funcția definită de utilizator, în fișierul *manipulareYaml.py*, utilizând funcția $find$ și $replace$ pentru a înlocui adresa din baza de cunoștințe cu un hyperlink prin comenzile HTML integrate în fișierul Python.

Totuși, scopul principal în discuția cu chatbot-ul este învățarea. Astfel, a fost creată o platformă de mesagerie în timp real destinată utilizatorilor eruditi. Pe această platformă vor discuta și își vor pune întrebări utilizatorii, pregătindu-se în acest mod pentru quiz-uri, teste sau diferite concursuri. Platforma de mesagerie este creată prin intermediul unui prize (socket), utilizând clasa *SocketIO* din librăria $flask\_socketio$, unde parametrul de *broadcast* este setat pe valoarea *adevărat*. Utilizatorii vor primii notificări când se va trimite un mesaj sau un fișier. Implementarea acestei secțiuni era îndeplinită în fișierul *service-worker.js* și *manifest.json*. Aceste două fișiere sunt conectate cu pagina principală de control al fluxului aplicației *default.html*.

Pentru a accesa chatbot-ul și ulterior și platforma *intellBot TK* utilizatorul va trebui să se înregistreze dacă nu are cont. Dacă are cont, va trebui să treacă prin partea de logare. Partea de înregistrare s-a realizat în fișierul *register.html*. Această pagină HTML este conectată cu calea creată în fișierul cu așa-numitele vederi. Este asociată funcției *register()*. Utilizatorul introduce datele sale personale, care-i permit să acceseze aplicația, în formele create în fișierul *forms.py*, utilizând librăriile `flask_wtf`, `wtforms` și modulul *validators* din `wtforms`. S-au creat clasele *RegisterForm* și ulterior *LoginForm* moștenind de la super-clasa *FlaskForm* importată. Clasele *RegisterForm* și *LoginForm* dispun de câteva câmpuri. Datele din aceste câmpuri sunt stocate în baza de date prin intermediul modelelor definite și dezvoltate în cadrul fișierului *models.py*. Tabela *user* reține datele introduse la înregistrare. Tabela este creată în fișierul *models.py* prin intermediul clasei declarate și definite de utilizator, *User*, care moștenește de la super-clasa importată, *UserMixin* și de la clasa *Model* din cadrul clasei *SQLAlchemy* care s-a importat din librăria `flask_sqlalchemy`. Toate obiectele claselor principale, ce sunt importate, au fost create și inițializate, în fișierul *__init__.py* din cadrul directoriului **app**, destinat aplicației. Tabela *User* este doar una din cele

patru tabele pe care le posedă baza de date creată după prima pornire a aplicației și configurată în fișierul *configuration.py*. Baza de date *SQLite* este denumită *erudite-messages.db* și pentru a găsi calea către ea, la partea de configurare s-a introdus calea absolută, după ce ea se salvează în directoriul principal **app**. Pe lângă inițializarea parametrului `SQLALCHEMY_DATABASE_URI` cu calea absolută către baza de date, a mai fost introdusă și o valoare pentru parametrul de setare a cheii secrete pentru a păstra confidențialitatea bazei de date.

După ce s-a logat, va avea posibilitatea să intre pe pagina de profil a utilizatorului, unde îi sunt prezentați chatboții cu care va avea un dialog, o discuție a două minți. În aplicație, care este una *responsive* (disponibilă și pe dispozitivele mobile), există și un *sidebar* de unde se pot selecta paginile (User Profile, Group Chat sau Logout).

3.2 Extinderea cazurilor de utilizare prin diagrama de secvență

În secțiunea de reprezentare mai amplă a cazurilor de utilizare se vor observa diagramele de secvență aferente acestor cazuri, cu scopul de a ilustra mai concret cititorului cele pomenite în secțiunea de descriere generală a problemei prin fluxuri de informații.

Actor: Programator
Nume Use Case: Documentare biblioteci și funcții
Descriere: Programatorul se documentează despre librăriile Flask, Scikit-learn, Natural Language Toolkit, Numpy, PyYaml și chatterbot și despre modulele, funcțiile și clasele de care dispun aceste librării. Anume, din librăria *nltk* pentru construirea conceptului de sistem expert s-au utilizat modulele *punkt*, *wordnet* și *popular*. Pentru a converti șirul de caractere în mai multe propoziții s-a utilizat metoda *sent_tokenize*, iar propozițiile în cuvinte metoda *word_tokenize* și s-a specificat linia de text la care aceasta se referă. Pentru a obține rădăcina cuvântului și pentru a executa detectarea aproximativă de referințe, s-a instanțiat un obiect de tip $WordNetLemmatizer$, iar apoi s-a găsit rădăcina cuvântului utilizând metoda *lemmatize*.

Clasa $TfidfVectorizer$ s-a utilizat pentru instanțierea unui obiect și pentru accesarea funcției $fit\_transform$ cu scopul de a obține forma obiectului aferentă cu cea necesară pentru utilizarea obiectului în cadrul funcției *cosine_similarity* drept parametru. Funcția *cosine_similarity* este esențială din cauza metodologiei de detecție a referinței în cazul în care același text coincide cu mai multe bibliografii. Atunci, construind o matrice, algoritmul implementat la fundalul acestei funcții selectează referința cea mai apropiată de perechea de indecși (0, 0). Următoarea fază care trebuie discutată și analizată este cea cu fișierele Yaml, care este una esențială în construirea acestui chatbot universal automatizat. S-a importat, pentru început, librăria $PyYaml$, iar ulterior s-au construi fișierele de tip Yaml, unde s-au inițializat parametrii cheie: *categories* și *conversations*, evident ținând cont de categorii (capitole) și perechile de întrebări și răspunsuri. După aceea, în fișierul *manipulareYaml*, *manipulareYamlRO* și *manipulareYAMLSRB* s-au construit algoritmi manuali pentru manipularea datelor, care ulterior au fost transferate în fișierele Python *functii*, *funcțiiRO* și *functiiSRB*. Ultimul pas al construirii lanțului erudit constă în importul fișierelor în fișierul principal, fișierul de bază, numit *views.py*. Scopul fișierului este să construiască vederi sau, așa-numitele rute (căi) pentru vizualizarea grafică. Totuși, după construirea rutei, se

va face o referință către pagina HTML sau JavaScript unde se vor efectua operații aferente tehnologiilor WEB, după transpunerea variabilelor în fișiere de acest tip după intercomunicarea cu fișierul *views.py*.
Condiție prealabilă: Actorul trebuie identificat în sistem ca și programator.

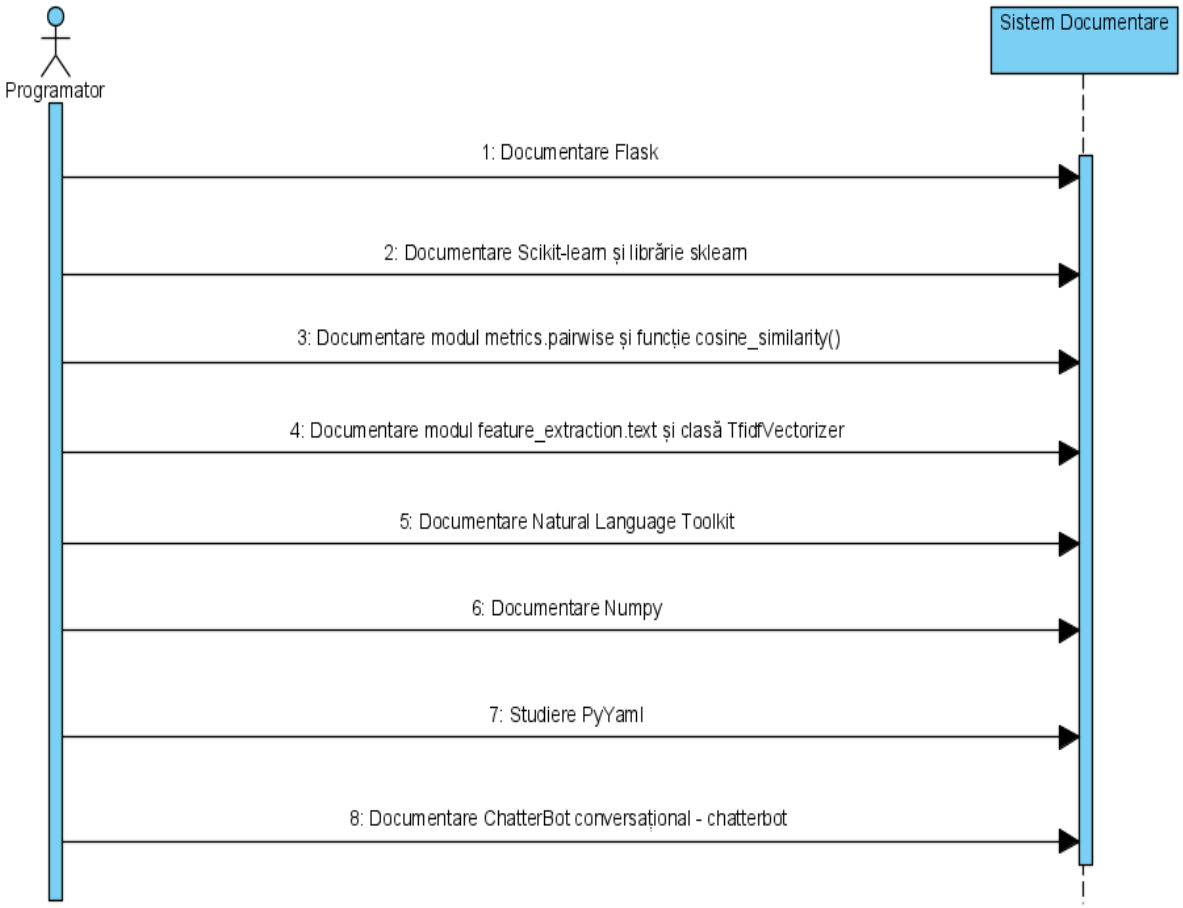

Figura 3.1: Partea de documentare

Actor: Programator
Nume Use Case: Dezvoltare aplicație software (Extragere/construirea datelor, proiectarea interfeței grafice)
Descriere: Specificarea modalităților de construire a datelor și de reprezentare pe interfața utilizatorului.
Condiție prealabilă: Actorul trebuie identificat în sistem ca și programator.

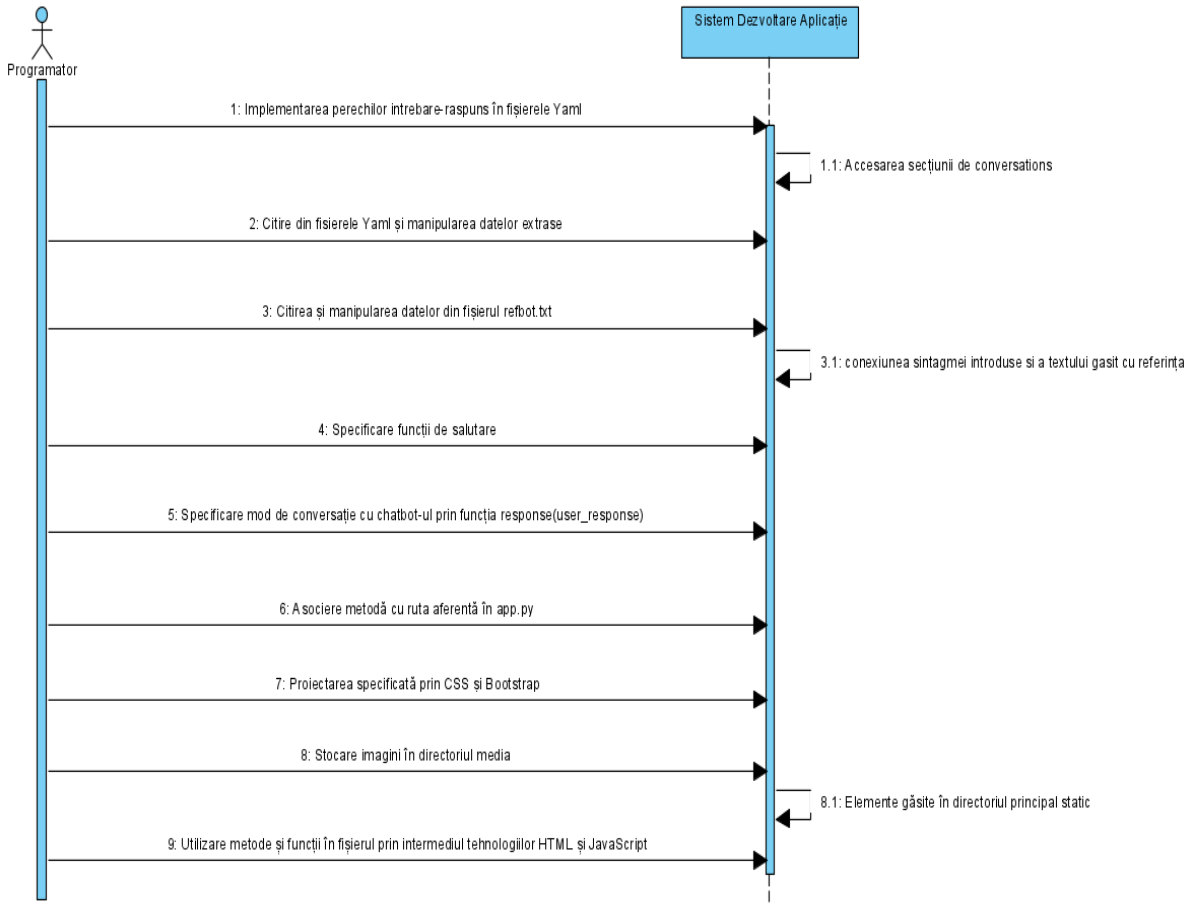

Figura 3.2: Dezvoltarea aplicației software

Actor: Programator
Nume Use Case: Structurare flux aplicație
Descriere: Reprezintă procesul de construire a etapelor de conversație cu chatbot-ul.
Condiție prealabilă: Actorul trebuie identificat în sistem ca și programator.

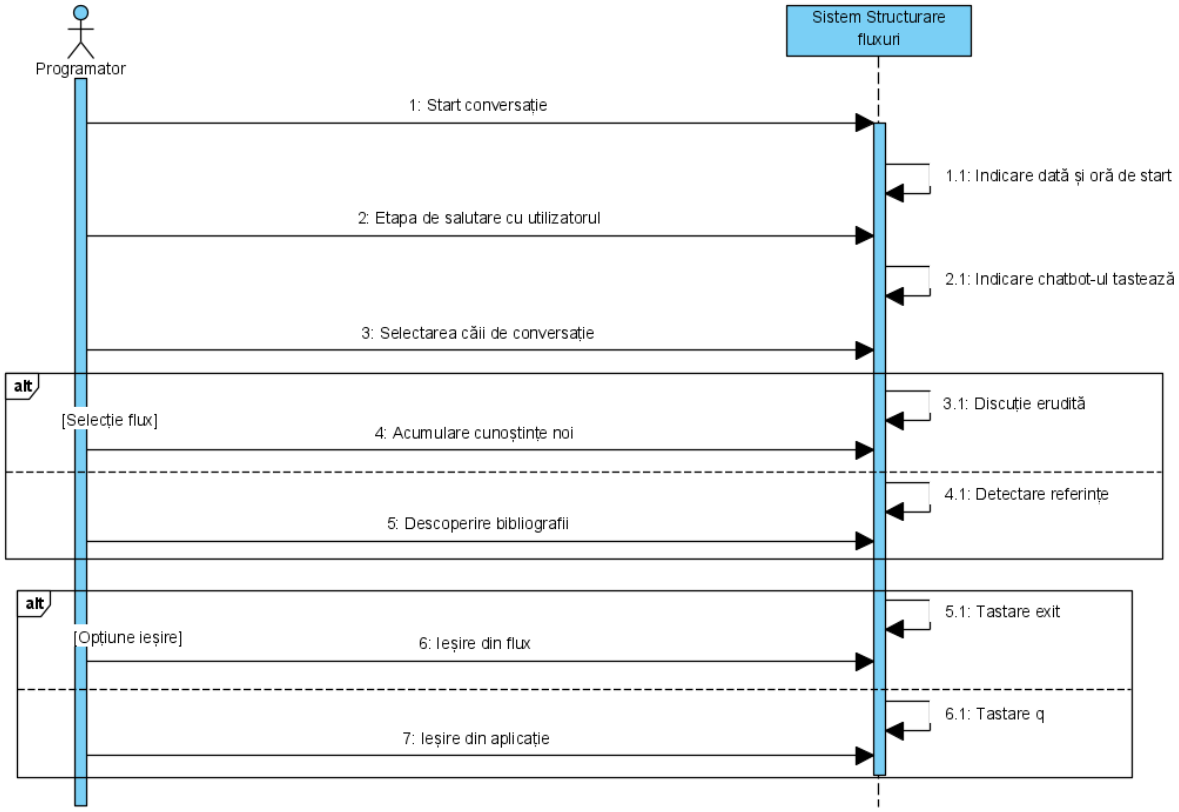

Figura 3.3: Prezentarea structurii fluxului aplicației

Actor: Programator
Nume Use Case: Mod realizare
Descriere: Aici se poate observa procesul de construire a rezultatelor de către programator.
Condiție prealabilă: Actorul trebuie identificat în sistem ca și programator.

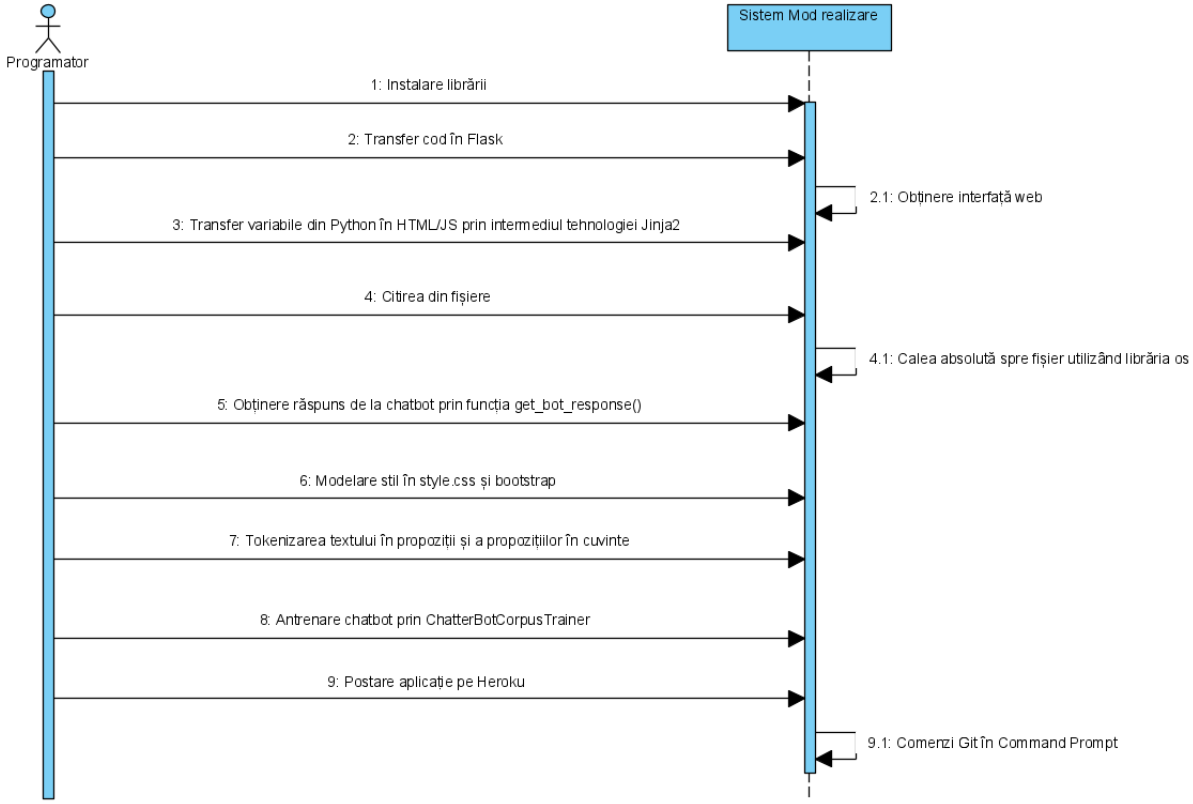

Figura 3.4: Principiile de realizare a aplicației

3.3 Fluxul aplicației

Această secțiune oferă o revizuire a fluxului aplicației prin specificarea acțiunilor îndeplinite în aplicație, utilizând diagrama de activitate.

După procesul de înregistrare sau logare la aplicație, utilizatorul are posibilitatea de a selecta limba în care dorește să discute cu chatbot-ul trilingv erudit și inteligent. Primul pas în discuția cu chatbot-ul o cuprinde partea de cunoaștere dintre chatbot și utilizator, în care chatbot-ul îl întreabă pe utilizator de poreclă, despre locul de muncă. Utilizatorul precizează locul de muncă al său, iar chatbot-ul, in conformitate cu butonașul ales, își construiește atitudinea de discuție cu utilizatorul. Următorul pas presupune vizualizarea confidenței utilizatorului deoarece acesta trebuie să zică, să confirme, dacă dorește să continue conversația cu această mașină virtuală, cu acest sistem expert. Dacă răspunsul este da, atunci se va selecta fluxul de discuție. Conversația poate fi redusă la detectarea referințelor sau poate fi transferată către secțiunea de pregătire pentru quizz-uri, jocuri de jeopardy, etc.

Următoarea secțiune este destinat utilizatorului. Anume, acesta va introduce un cuvânt, o sintagmă, sau va pune o întrebare și se va forma un dialog, o convorbire a celor două minți, una umană, iar cealaltă automatizată, virtuală. Totuși, dacă utilizatorul alege secțiunea de detectare a referințelor, atunci după un anumit text introdus de utilizator, chatbot-ul va specifica referința concretă, precisă, aferentă textului, sau referința apropiată textului introdus. Pentru a termina conversația cu chatbot-ul se tastează *exit*.

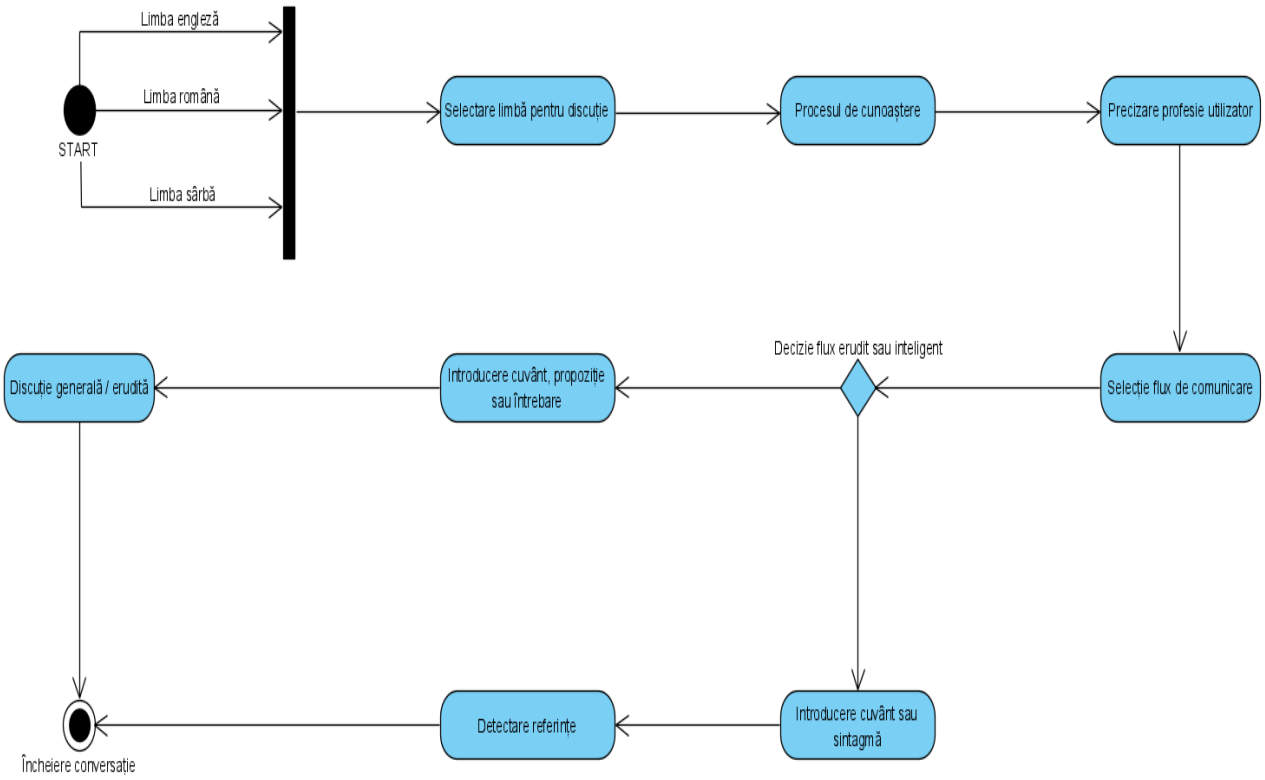

Figura 3.5: Fluxul aplicației utilizând diagrama de activitate

3.4 Reprezentarea grafică a soluției

Această secțiune este destinată reprezentării grafice a soluției propuse integral. În prima imagine se poate observa pagina de înregistrare în care utilizatorul a uitat să-și introducă numele de familie. Acest lucru este semnalat prin mesajul: *Please fill out this field*. În imagine se poate de asemenea să observăm un buton pentru pornirea locației ca și element de Progressive Web Application numit *Geolocation*. Mai se poate lua la seamă și posibilitatea de a accesa pagina de logare, apăsând pe *Login*. Și footer-ul este vizibil în josul paginii.

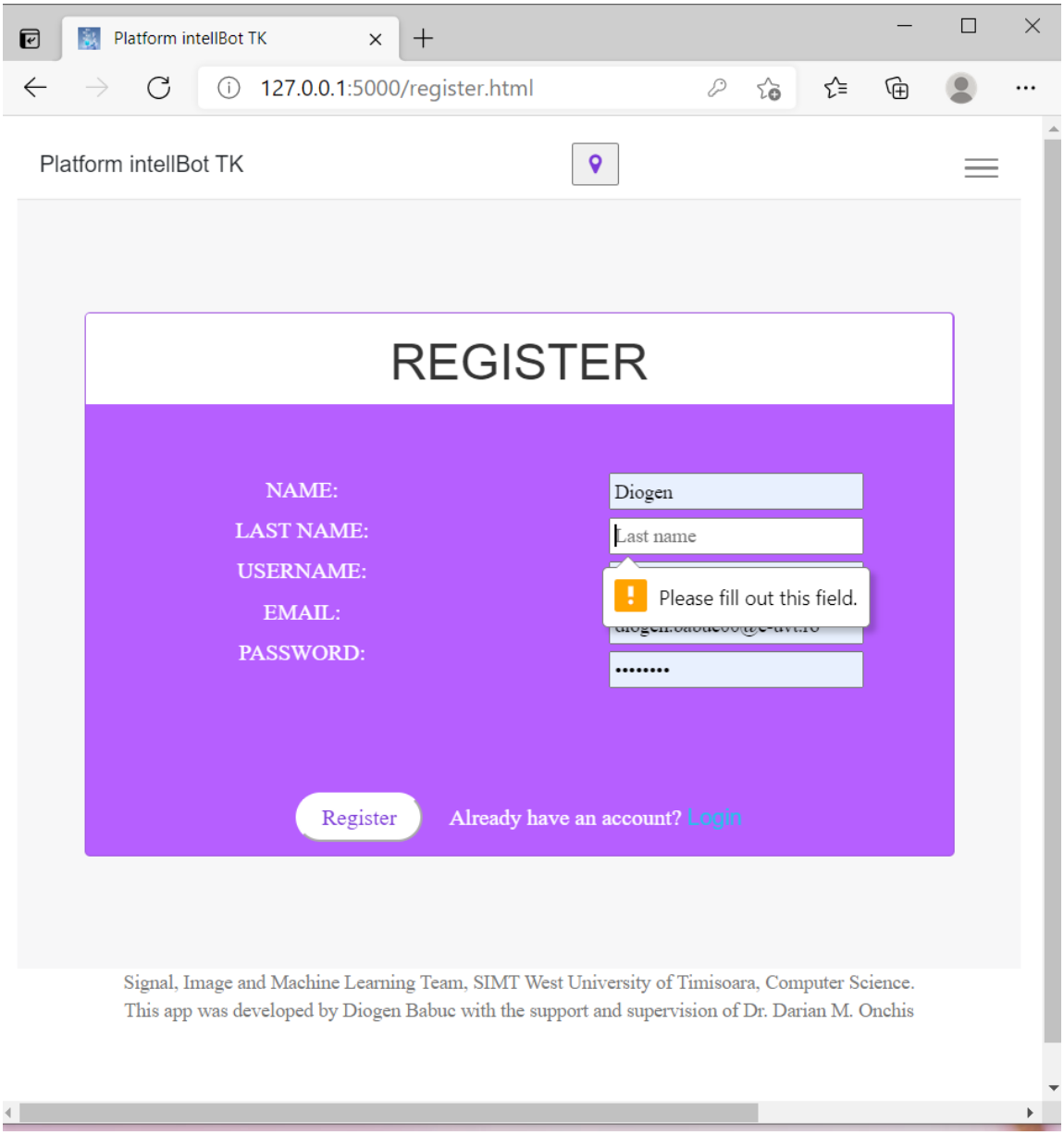

Figura 3.6: Pagina de înregistrare

Mai departe se poate vizualiza pagina de logare, în care utilizatorul a introdus un nume de utilizator incorect și inexistent în baza de date. Lucrul respectiv este semnalat printr-un mesaj de tip notificare. Încă o situație care ar fi fost semnalată apărea atunci când parola nu s-ar fi potrivit cu numele de utilizator introdus în câmpul care sugerează aceasta.

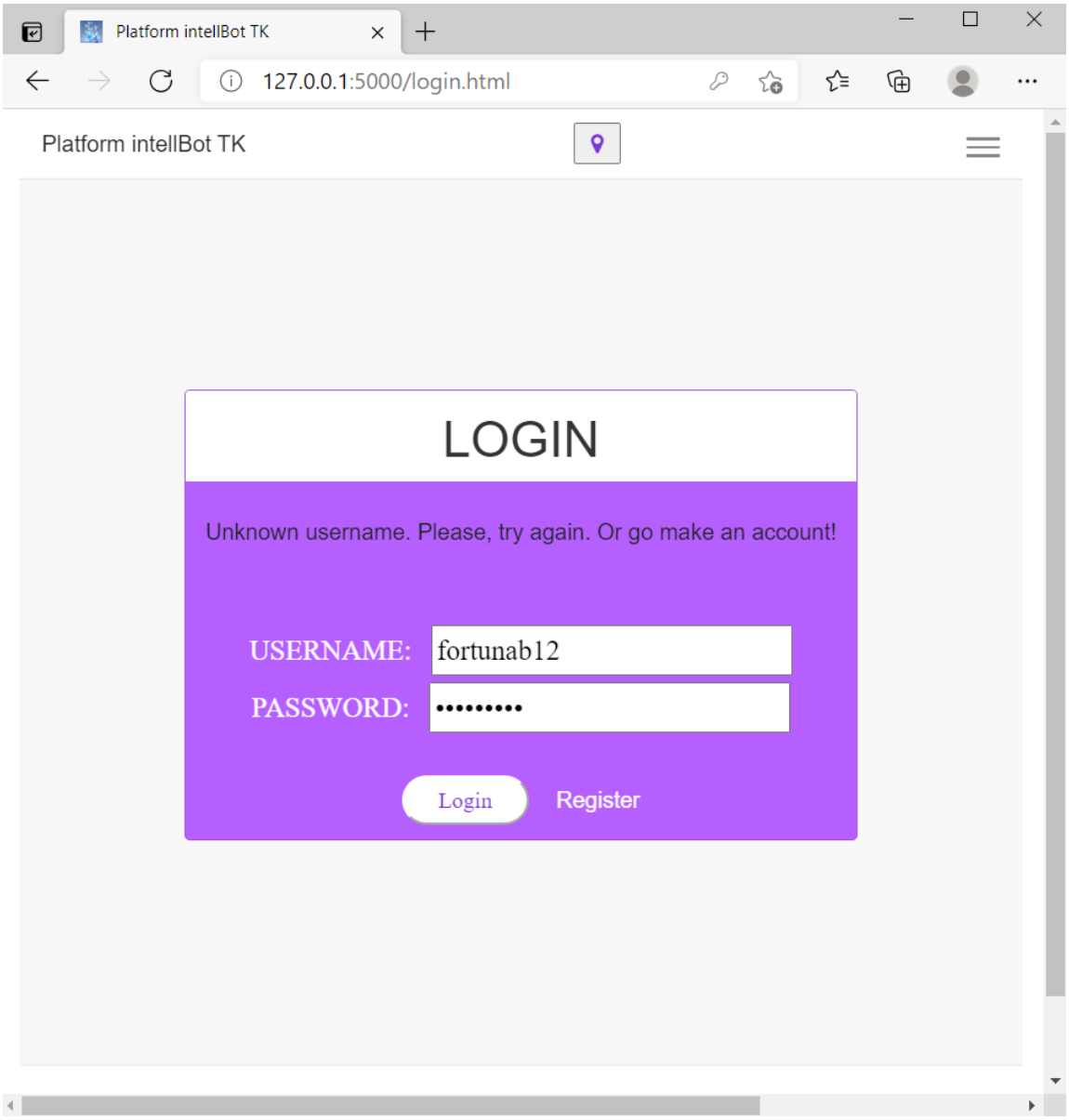

Figura 3.7: Pagina de logare

Prima pagină pe care utilizatorul o accesează după ce a intrat în aplicație este pagina de profil al utilizatorului. Utilizatorul își poate preciza sexul după ce selectează imaginea respectivă ca și imagine de profil. Are trei opțiuni de discuție cu chatbot-ul: să discute în limba română, în limba sârbă sau în limba engleză, astfel încât să fie pregătit pentru conversațiile ulterioare cu alte ființe umane pe platforma *intellBot TK* care se desfășoară în timp real. Totuși, platforma, ca și întreaga aplicație, sunt disponibile și în regimul off-line din cauza elementului de *Progressive Web Application*, *Background Sync*, utilizat. De asemenea, aplicația este una responsive, adică disponibilă și pe dispozitivele mobile care accesează aplicația pe Internet.

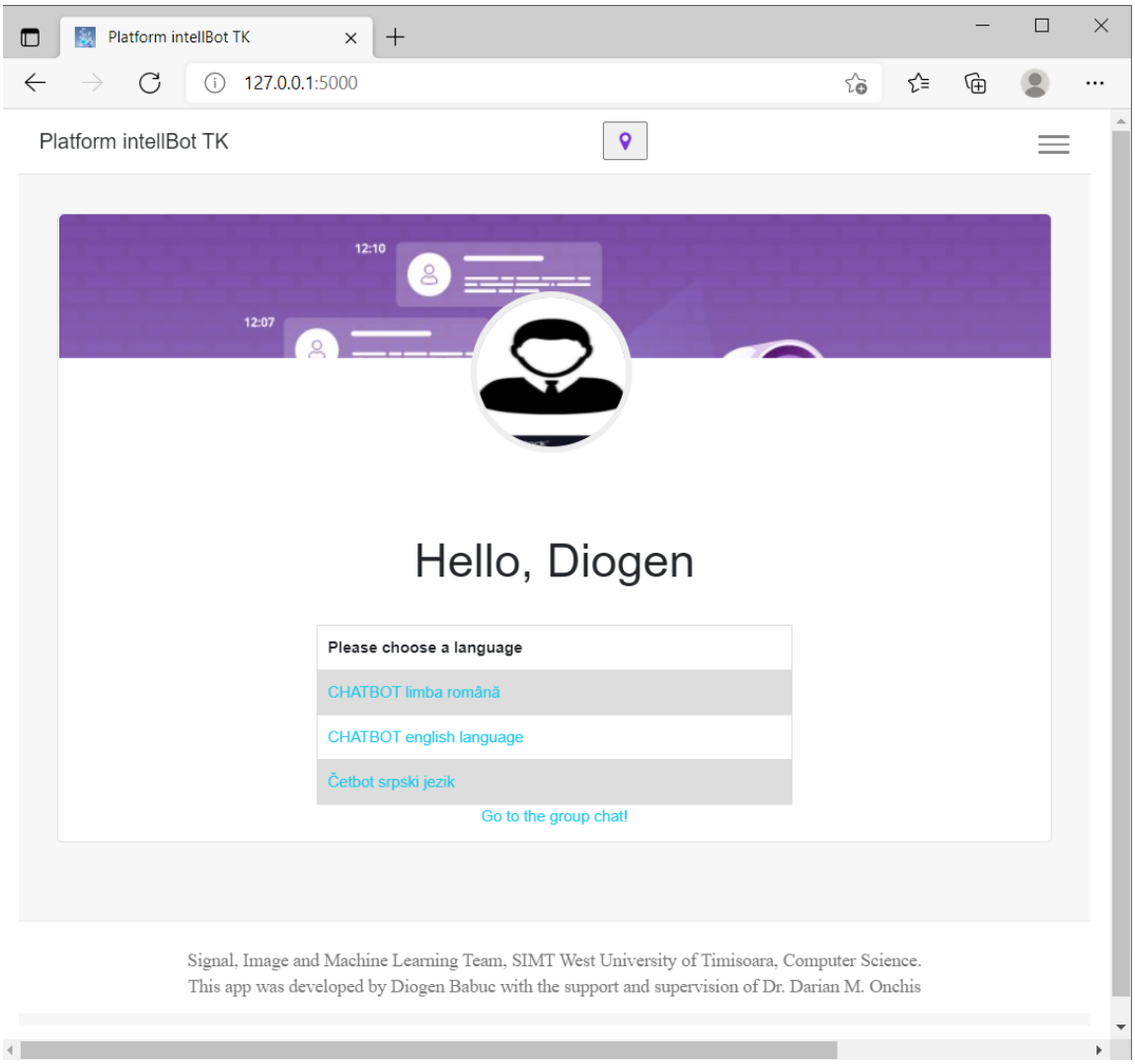

Figura 3.8: Pagina de profil a utilizatorului

Pagina *englishlang.html* este menită discuției cu chatbot-ul numit XAIBOT în limba engleză. Ca și o introducere în aplicație și ca un motiv de încunoștințare a utilizatorului, pe pagină există și o scurtă descriere a platformei pentru comunicarea mai multor utilizatori în timp real. Este vizibil momentul când chatbot-ul începe să tasteze un mesaj. Utilizatorul are posibilitatea de a tasta și de a învăța ceva nou de la chatbot, sau de a-l învăța pe chatbot.

De asemenea, în imagine mai se poate observa și că utilizatorul și-a activat locația.

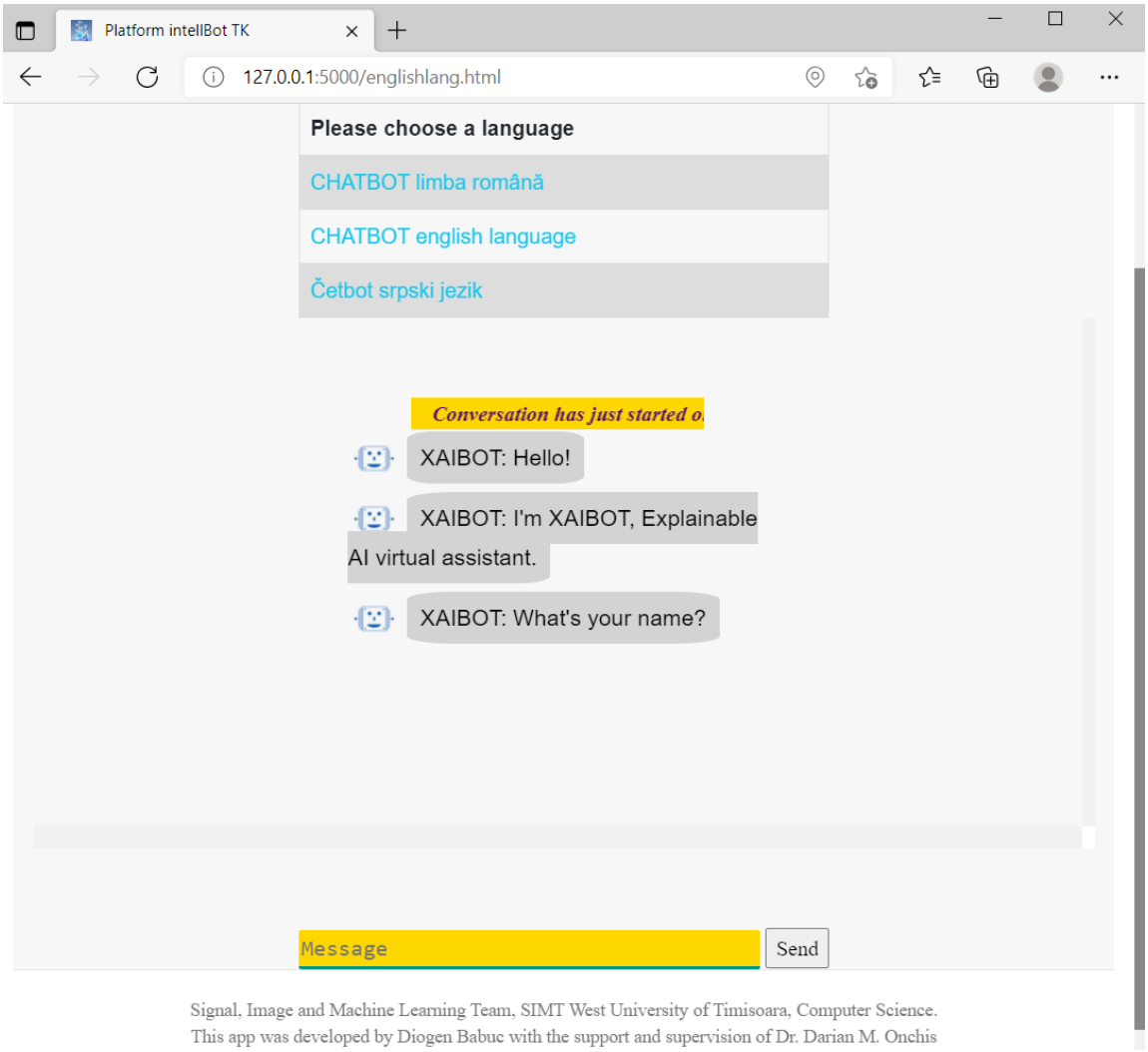

Figura 3.9: Început de conversație cu chatbot-ul în limba engleză

Pagina de mesagerie în timp real se poate vizualiza în imaginea de mai jos. Anume, n utilizatori au accesat aplicația și au început discuția, dialogul. Își pun întrebări, răspund la ele, se corectează reciproc dacă apar greșeli. Pe lângă numele de utilizator și mesaje, apare și timpul când un mesaj sau fișier a fost trimis. Trimiterea fișierelor se desfășoară prin intermediul elementului de aplicație web progresivă numit *Hardware accesul fișierelor*. După ce un mesaj a fost trimis sau un fișier a fost încărcat, va apărea notificarea de tip *împinge* (push) care va semnala faptul că a fost adăugat text pe platformă.

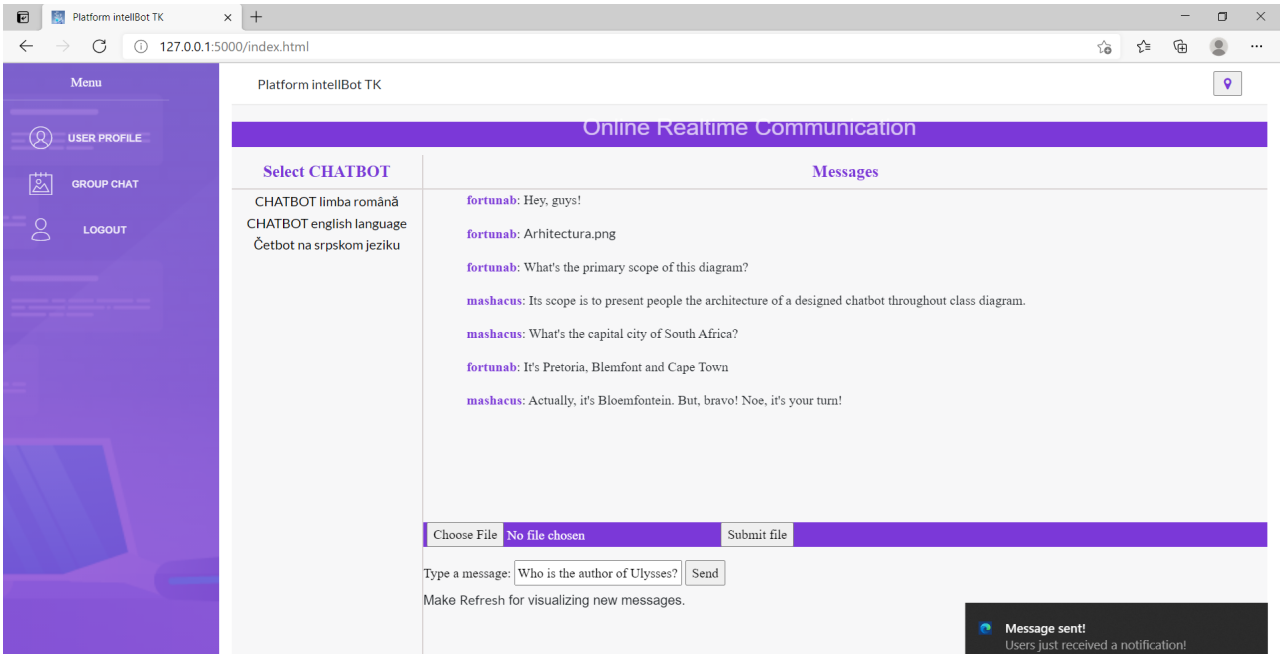

Figura 3.10: Platforma *intellBot TK* pentru comunicarea online și offline

Capitolul 4

Detalii de dezvoltare și implementare

Capitolul cu detalii despre implementare presupune definirea unor limbaje de programare sau tehnologii recent utilizate de autor care nu se folosesc la o scară largă, precum și specificarea funcțiilor, metodelor, claselor și a principiilor folosite în aplicație. Ulterior, se va menționa locul și modul de utilizare a obiectelor în aplicație. O să observăm și care este contribuția autorului asupra codului și asupra întregii organizații de modelare, dezvoltare și implementare a aplicației.

4.1 Tehnologii folosite

Tehnologiile care se utilizează sunt: Python, Flask, Jinja2, HTML, CSS (inclusiv Bootstrap), JavaScript (inclusiv JQuery), *WebView* și baza de date (SQLite).

Flask este un cadru de lucru pentru aplicațiile web, care conectează partea de back-end cu cea de front-end, deci conectează partea de cod scrisă în tehnologiile web cu secvențele de cod scrise în Python. Pentru a utiliza acest cadru de lucru este necesar să se importă librăria $flask$ în Python. Pentru implementarea în Python și tehnologiile web se folosește mediul de dezvoltare *PyCharm*. $Jinja$ este un limbaj modern destinat pentru modelarea în limbajul de programare Python. Are o structură asemănătoare cu șabloanele $Django$. Este rapid și de aceea utilizat pe o scară largă.

Pentru a explora datele și pentru a le manipula, s-a folosit limbajul de programare Python, împreună cu cadrul de lucru Flask, pe când pentru reprezentarea grafică se utilizează HTML, pentru partea de proiectare CSS, iar pentru funcționalități JavaScript. *JQuery* permite efectuarea conexiunii dintre Python și JavaScript prin intermediul rutei care se indică drept parametru al funcției *get*. Utilizând limbajul de programare Python, se configurează baza de date, se construiesc modelele aferente tabelelor, coloanelor și operațiilor posibile asupra acestora, cu includerea constrângerilor. Formele în care utilizatorul își introduce datele personale sunt construite în Python, moștenind din clasa *FlaskForm*. Câmpul pentru trimiterea mesajelor sau fișierelor este la fel văzut drept formă.

4.2 Contribuție asupra dezvoltării și implementării codului

În secțiunea menită pentru specificarea contribuțiilor personale asupra dezvoltării și implementării codului se va vedea unde și în ce mod s-au utilizat tehnologiile pomenite în secțiunea precedentă, pentru a obține un rezultat. Se va mai preciza cum a fost realizată lucrarea pe exemple concrete și se va specifica structura generală a aplicației împreună cu modurile pentru modelare și funcționare.

Pentru a obține o interfața grafică, web, astfel încât să se poate comunica cu chatbot-ul și pe Internet, s-a utilizat cadrul de lucru pentru aplicațiile micro-web transcris în limbajul de programare Python, numit *Flask*. De asemenea, s-a folosit și șablonul *Jinja* pentru transferul de variabile din Python în JavaScript sau HTML. Eventual s-ar putea folosi și la păstrarea unui istoric al conversațiilor dintre XAI-BOT și utilizator, utilizând o bază de date, de pildă *SQLite*. Se utilizează funcția `download()` din librăria *nltk* pentru a descărca unele pachete, cum ar fi *punkt* sau *worknet* [BKL19]. Apoi, se citește din fișierul text cu *with open()*. Se folosește funcția `lower()` pentru a transforma toate literele majuscule în minuscule. Un progres care a fost îndeplinit a fost posibilitatea asistentului virtual de a citi cuvântul, fie el scris cu litere mari, fie cu litere mici. În proiect există și două liste cu șirurile de caractere posibile pentru salutarea cu chatbot-ul. O parte importantă la citirea din fișierul text este tokenizarea, adică divizarea textului în propoziții cu `sent_tokenize()`, respectiv propozițiilor în cuvinte cu `word_tokenize()` [BKL19] din librăria nltk. Răspunsul generat sau returnat de asistentul virtual este oferit de funcția `response()`. Dacă utilizatorul introduce drept date de intrare cuvântul *exit*, atunci se iese din program, iar chatbot-ul se salută cu utilizatorul.

S-a importat și instalat flask-ul și alte librării necesare pentru program și s-a creat o variabilă de tipul clasei Flask, cu parametrul `name`. La citirea din fișiere s-a utilizat calea absolută spre fișier, prin funcția:

`os.path.dirname(os.path.abspath(__file__)`.

Etapa de salutare cu botul a fost introdusă direct în funcția `response(user_response)` deoarece funcția cu numele `get_bot_response()` returnează chiar funcția de răspuns. Pentru această funcție s-a creat calea *get*. În funcție se preia textul introdus în câmpul de text pentru transmiterea mesajelor. Mesajul introdus pe interfața de utilizator se transferă în Flask prin tehnologia *Jinja2*. În fișierul *index.html*, crearea și utilizarea funcției `getBotResponse()` permite lui XAIBOT să găsească propoziția și referința aferentă cuvântului introdus la input. Fișierul *index.html* din directoriul templates permite vizualizarea interfaței grafice. Stilul este modelat în fișierul *style.css* din directoriul static. La rularea aplicației, se crează automat baza de date în *SQLite3* și fișierul cu extensia pickle și numele *sentence_tokenizer*, care tokenizează textul din fișierul text *refbot.txt* în propoziții, iar propozițiile în cuvinte.

Proprietățile secțiunii de detectare a citatului sunt declarate în fișierul *chatterREF.py*. Este importată biblioteca *os* pentru a găsi calea directoarelor, iar în directoare și calea absolută spre fișierele dorite (refbot.txt și referinte.txt), și pentru a citi din aceste fișiere. După citire, s-a făcut conversia din lista cu un singur text amplu, în lista cu propoziții, prin `sent_tokenize(linie)` [BKL19] din biblioteca importată, nltk. De asemenea, pentru a grupa diferite forme de cuvinte astfel încât ele să poată fi ușor de analizat, se utilizează clasa *WordNetLemmatizer* din pachetul *nltk.stem*

[Raj19]. Funcția principală este `response(raspunsul_utilizatorului)` definită de programator, unde se găsește și funcția pentru salutarea chatbot-ului cu utilizatorul. Procesul aplicat pentru detectarea propoziției în care se găsește cuvântul/sintagma introdusă este `cosine_similarity()` din modulul *metrics.pairwise* din *sklearn* [Joh19].

Într-o variabilă se păstrează textul introdus de utilizator prin comanda:

`request.args.get('msg')`,

iar pentru a porni conversația în limba engleză discutând despre diferite domenii, se accesează metoda `get_response()` de instanța clasei ChatBot, iar textul accesat din interfață este introdus drept parametru. În cazul în care textul introdus de utilizator este *exit*, atunci se iese din flux. În fișierul *.html*, ruta */get* se utilizează în funcția:

`discutie_noua()`,

accesându-se cu metoda *get* din *JQuery*, iar mesajul transmis de utilizator se accesează prin intermediul tehnologiei *Jinja2*, utilizând comanda {`msg:rawText`}, unde *msg* este mesajul utilizatorului, iar *rawText* variabila în care se păstrează textul. Un întreg ciclus al conversației a fost descris prin funcțiile definite în Javascript sau rutele asociate metodelor din Flask/Python.

Pentru a completa cunoștințele chatbot-ului, s-au adăugat informații din domeniul XAI, AI, al calculatoarelor și al rațiunii botului în fișierul yaml. Din fișierul yaml s-a citit cu ajutorul comenzii `with open(r'xai.yml')`, după ce s-a importat biblioteca yaml. Pentru a încărca textul din fișierul yaml, s-a folosit metoda `load()`, unde s-a introdus variabila pentru fișier și atributul *loader=yaml.FullLoader*. Pentru a accesa secțiunea de conversații, deoarece datele au fost stocate în variabila `xai_list` care este dicționar, era nevoie să se parcurgă lista: *xai_list['conversations']*. Pentru a plasa datele din fișierul yaml pe website, s-a construit ruta */get* pentru funcția definită de programator, numită `get_bot_response()`. Funcția care s-a accesat la returnare este funcția *convorbireXAI* din fișierul manipulareYaml. Pentru a extinde textul pentru detectarea referințelor, s-a adăugat text în fișierul refbot.txt, din articolul *Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI*. Pentru a plasa aplicația pe Heroku, trebuia creat un fișier Procfile cu textul: *web: gunicorn app:app*, un fișier nltk.txt în care au fost introduse pachetele necesare de care s-a folosit biblioteca nltk în program (punkt, wordnet și popular), un fișier requirements.txt în care s-au introdus toate bibliotecile pe care le-a utilizat aplicația (flask, sklearn, nltk, chatterbot, gunicorn), astfel încât bibliotecile externe să se instaleze sau descarce automat. Pentru a pune site-ul pe Heroku, trebuiau să se tasteze câteva comenzi în Command Prompt. Anume: *git init, heroku git:remote -a nume_depozit_app_heroku, git add ., git commit -am „make it better" și git push heroku master*. Se va crea în librăria rădăcină un directoriu numit Lib, unde se vor stoca pachetele, bibliotecile și funcțiile instalate sau descărcate. Trebuia și să existe grijă ca în directorul rădăcină să se creeze fișierul *pyvenv.cfg*. În directoriul *Scripts* s-au creat câteva executabile, într-un mod automat, pentru ca programul să poată rula. Pentru a structura aplicația, s-a folosit un bootstrap și clasele *container-fluid* și *align-contentcenter* din acel bootstrap. Pentru a plasa codul pe GitHub, s-a creat un depozit, numit *XAIBOT-Virtual-Assistant*, iar fișierele au fost introduse în depozit prin *drag and drop*.

4.3 Cod

În această secțiune se va reprezenta și analiza codul implementat. De asemenea, se vor indica modalitățile de implementare. Codul integral al aplicației urmează să se afle pe Github, la adresa: `https://github.com/fortunab/LicentaTestare`.

După implementarea tuturor claselor necesare din librăriile aferente lor, am creat obiecte de tipul acestor clase. Obiectul de tip *Flask* se folosește la crearea căilor pentru site-ul web și la configurarea bazei de date, pentru care s-a declarat un obiect de tip *SQLAlchemy*. S-au creat și un obiect pentru criptarea parolei și unul esențial pentru gestionarea părții de logare. S-a construit și o metodă care creează baza de date după prima rulare a aplicației, prin intermediul funcției *create_all()*. Ulterior se importează fișierele care dețin vederile și modelele aplicației, definite de programator.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager
from flask_bcrypt import Bcrypt


app = Flask(__name__)
app.config.from_object('app.configuration.Config')
db = SQLAlchemy(app)
bc = Bcrypt(app)
lm = LoginManager()
lm.init_app(app)

@app.before_first_request
def initialize_database():
    db.create_all()

from app import views, models
```

A doua secțiune importantă a implementării, luând-o în ordine, apare în fișierul destinat configurării bazei de date, de unde am extras doar unul din cei patru parametrii. Acest parametru este *SQLALCHEMY_DATABASE_URI* și el specifică faptul că baza de date, ce se va crea, este *SQLite*, iar apoi s-au specificat directoriul de bază, utilizând calea absolută care preia rularea scriptului, și numele bazei de date.

```
import os

basedir = os.path.abspath(os.path.dirname(__file__))
SQLALCHEMY_DATABASE_URI =
    'sqlite:///' + os.path.join(basedir, 'eruditemessages.db')
```

Partea de creare a modelelor este una foarte importantă în construirea tabelelor și coloanelor din baza de date. Dintre cele patru tabele (user, friend, message și files), ca și exemplificare și ilustrare a fost selectată tabela *user*, care este una inevitabilă la partea de construire a secțiunii de înregistrare în aplicație.

Anume, s-a implementat clasa, moștenind de la clasa predefinită din *flask_login*, denumită *UserMixin* și moștenind de la clasa *Model* care se accesează prin prezența obiectului de tip *SQLAlchemy*. Denumirea tabelei se realizează folosind parametrul predefinit al clasei, *__tablename__*. Următorul pas este crearea coloanelor și setarea proprietăților, unde se enumeră și specificarea tipului de date, și caracteristicile de accesibilitate, și partea relațională a bazei de date relaționale. Coloanele se definesc prin intermediul clasei *Column*, accesată din *SQLAlchemy*. Ulterior se va inițializa

constructorul clasei, modul de reprezentare principal și metoda *save()* care dispune de funcțiile predefinite, de *add()* și *commit()* sau *rollback*, în conformitate cu integritatea înregistrării datelor. La partea de modelare a bazei de date a existat și o metodă pentru eliminarea înregistrărilor din tabela cutare, însă ea a fost eliminată din întrebuințare datorită faptului că utilizatorul nu mai poate nici adăuga, nici șterge prieteni din lista sa de prieteni. După evidențierea celor pomenite anterior, de menționat este că tabela *friend* din baza de date este goală, însă nu a fost eliminată din secțiunea de modelare a bazei datorită direcțiilor viitoare, care presupun posibilitatea utilizatorilor să inter-comunice în chat-ul privat.

```
from app import db
from flask_login import UserMixin
from sqlalchemy.exc import IntegrityError

class User(UserMixin, db.Model):
    __tablename__ = 'user'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), nullable=False)
    lastn = db.Column(db.String(64), nullable=False)
    username = db.Column(db.String(64), unique=True, nullable=False)
    email = db.Column(db.String(60), unique=True, nullable=False)
    password = db.Column(db.String(50), nullable=False)

    def __init__(self, name, lastn, username, email, password):
        self.name = name
        self.lastn = lastn
        self.username = username
        self.email = email
        self.password = password

    def __repr__(self):
        return '<User %r - %s>' % (self.id) % (self.email)

    def save(self):
        db.session.add(self)
        try:
            db.session.commit()
        except IntegrityError:
            db.session.rollback()
        return self
```

Deoarece fișierul *forms.py* nu este unul cu multe linii de cod, s-a decis să fie integral introdus în această secțiune. Anume, după ce au fost importate clasele necesare, printre care cea mai importantă este *FlaskForm* care constituite o super-clasă pentru fiecare clasă definită manual în acest fișier, s-au indicat proprietățile câmpurilor pe care le va accesa și pe care le va completa utilizatorul. Se poate observa ușor căci fiecare model definit anterior are asociat o formă, prin intermediul căreia se specifică textul ce urmează a fi adăugat în tabelele bazei de date, executând metoda *save()* după declararea unui obiect de tipul clasei create în *models.py*. Și în fișierul cu formele, precum și în cel cu modele, se indică validările ce urmează a fi efectuate. Validările, în general, se săvârșesc pentru adresa de email, parolă și nume de utilizator, tratând doar secțiunea de *RegisterForm*. Din codul introdus se poate observa ușor care sunt validările implementate asociate fiecărei clase din scriptul cu forme. Validările s-au importat din modulul *validators* care se găsește în librăria `wtforms`.

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField
from wtforms.validators import Email, DataRequired, Length


class LoginForm(FlaskForm):
  username = StringField(u'Username', validators=[DataRequired(
   message='Parola sau nume de utilizator incorecte.')])
  password = PasswordField(u'Password', validators=[DataRequired(
   message='Parola sau nume de utilizator incorecte.')])

class RegisterForm(FlaskForm):
  name = StringField(u'Name', validators=[DataRequired()])
  lastn = StringField(u'Lastn', validators=[DataRequired()])
  username = StringField(u'Username', validators=[DataRequired(
   message='Enter a unique username.')])
  email = StringField(u'Email', validators=[DataRequired(), Email(
   message='Enter a valid email.')])
  password = PasswordField(u'Password', validators=[DataRequired(),
   Length(min=6, message='Select a stronger password.')])

class FriendForm(FlaskForm):
  name_fr = StringField(u'NameFr', validators=[DataRequired()])
  username_friend = StringField(u'username_friend', validators=[
   DataRequired()])
  username_fk = StringField(u'username_fk', validators=[DataRequired
   ()])

class MessageForm(FlaskForm):
  username_mes = StringField(u'username_mes', validators=[
   DataRequired()])
  text_mes = StringField(u'text_mes', validators=[DataRequired()])
```

Partea centrală a implementării, care controlează și leagă scriptele de care dispune aplicația aceasta, o reprezintă fișierul cu vederi, destinat în principal creării de căi web, care vor permite vizualizări de pagini multiple, a căror adresă web se va distinge reciproc. Pentru secțiunea de logare și secțiunea de înregistrare în aplicație s-au construit funcții, cu nume distincte, s-au specificat rutele ce vor fi adăugate după adresa principală web și port, s-au indicat metodele de solicitare (cerere) HTTP deasupra funcțiilor și s-a pornit cu implementarea. La ambele părți s-a verificat, mai întâi, dacă utilizator are un cont. Dacă nu are un cont, la partea de înregistrare s-a inițializat forma aferentă, creată ca și clasă în fișierul *forms.py* și s-a indicat ca și atribut al clasei forma solicitată din cerere. Procesul se execută datorită metodei de cerere HTTP, *post*. Dacă acea formă s-a obținut aproape instant, ceea ce înseamnă că datele au fost introduse de utilizator în formele de la interfața web, se verifică existența metodei de cerere HTTP, *get*, pentru a obține datele și a lansa pagina web care urmează după înregistrare. Dacă apare vreo excepție din cauza integrității datelor sau a nerespectării constrângerilor, se va semnala, prin intermediul variabile *msg*, nepotrivirea datelor introduse cu cele solicitate, afișându-se mesajul aferent erorii de integritate care apare. Pentru partea de logare se vor trata doar coloanele în care sunt stocate parolele și numele de utilizator, respectând inevitabil unicitatea și potrivirea datelor introduse și la această secțiune. După validarea datelor introduse la logare, se conectează funcția din *flask* cu pagina html prin intermediul funcției importate predefinite, *render_template()*, specificându-se directoriul în care se află fișierul și conținutul fișierului în sine care urmează să fie

manipulat sau executat. Pentru a traversa de la pagina de logare la pagina de profil a utilizatorului, deoarece aceasta urmează, se utilizează randarea șablonului cu atributul care specifică calea relativă către fișierul care distribuie rolurile la pagini, *default.html* din directoriul *layouts*. De asemenea, încă o funcție declarată și definită de utilizator esențială în realizarea aplicației este cea în care se încarcă utilizatorul după datele introduse în forme (celule de la interfața web) și i se atribuie un număr de identificare ordinal.

```
@lm.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

@app.route('/logout.html')
def logout():
    logout_user()
    return redirect(url_for('login'))

@app.route('/register.html', methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('register'))
    #Declaram forma pentru inregistrare
    msg = None
    form = RegisterForm(request.form)
    if request.method == 'GET':
        return render_template('layouts/default.html', content=
   render_template('auth/register.html', form=form, msg=msg))

    msg = 'You have register succesfully'
    # Verificam daca forma este valida
    if form.validate_on_submit():
        #Asignam forma datelor variabile
        name = request.form.get('name', '', type=str)
        lastn = request.form.get('lastn', '', type=str)
        username = request.form.get('username', '', type=str)
        email = request.form.get('email', '', type=str)
        password = request.form.get('password', '', type=str)
        #users = User.query.all()
        #server = Server.query.all()
        user1_mail = User.query.filter_by(email=email).first()
        user1_username = User.query.filter_by(username=username).
   first()
        #user1_passw = User.query.filter_by(password=password).first
   ()
        if user1_mail:
            msg = 'This email is not available. Please, try with
   another one!'
        if user1_username:
            msg = "Username is not available. Please, try again!"

        pw_hash = password
        user = User(name, lastn, username, email, pw_hash)
        user.save()
    else:
        msg = 'Weak password. Try with a longer one (min. 6)'
    return render_template('layouts/default.html', content=
   render_template('auth/register.html', form=form, msg=msg))
```

```
45
46 # Autentificare
47 @app.route('/login.html', methods=['GET', 'POST'])
48 def login():
49     if current_user.is_authenticated:
50         return redirect(url_for('userprofile'))
51     form = LoginForm(request.form)
52     # Mesaje in cazuri de eroare
53     msg = None
54     # Verificam daca POST si forma sunt valide la submitere
55     if request.method == 'GET':
56         return render_template('layouts/default.html', content=
    render_template('auth/login.html', form=form, msg=msg))
57     if form.validate_on_submit():
58         username = request.form.get('username', '', type=str)
59         password = request.form.get('password', '', type=str)
60         user = User.query.filter_by(username=username).first() #
    preluam user-ul dupa username sau
61         if user:
62             if user.password == password:
63                 login_user(user)
64                 return redirect(url_for('userprofile'))
65             else:
66                 msg = 'Wrong password. Please, try again!'
67         else:
68             msg = "Unknown username. Please, try again. Or go make an
     account!"
69     return render_template('layouts/default.html', content=
    render_template('auth/login.html', form=form, msg=msg, password=
    password)) #default distribuie roluri la pagini
70
71 #ruta principala
72 @app.route('/', methods=['GET', 'POST'])
73 def userprofile():
74     if not current_user.is_authenticated:
75         return redirect(url_for('login'))
76
77     msg=None
78     form = FriendForm(request.form)
79     formr = RegisterForm(request.form)
80     user = User.query.all()
81     server = Friend.query.all()
82     if request.method == 'GET':
83         return render_template('layouts/default.html', content=
    render_template('pages/user.html', server=server,
84
              user=user, form=form, formr=formr, msg=msg))
85
86     user_username = [item.username for item in User.query.distinct(
    User.username).all()]
87     print(user_username)
88     if form.validate_on_submit:
89         try:
90             name_fr = request.form.get('name_fr', '', type=str)
91             username_friend = request.form.get('username_friend', '',
     type=str)
92             print(username_friend)
93             username_fk = request.form.get('username_fk', '', type=
```

```
    str)
94              server1 = Friend(name_fr, username_friend, username_fk)
95              server1.save1()
96              userfr_nume = Friend.query.filter_by(username_friend=
    username_friend).first()
97          except Exception:
98              return render_template('layouts/default.html', content=
    render_template('pages/user.html', server=server,
99
                    username_friend=username_friend, user=user, form=
    form, formr=formr, msg=msg))
100
101     return render_template('layouts/default.html', content=
    render_template('pages/user.html', server=server,
102
          username_friend=username_friend, user=user, form=form, formr
    =formr, msg=msg))
```

O aplicație web progresivă oferă funcționalități esențiale și unice în creare și dezvoltarea unei aplicații web native. De aceea, s-au utilizat trei elemente principale din secțiunea de PWA (Progressive Web Application). Primul element introdus, și cel mai important, presupune crearea și trimiterea notificărilor după trimiterea unui mesaj sau fișier către un utilizator sau către toți utilizatorii implicați pe platforma de comunicare în timp real, și online, și offline. În fișierul *app.js*, care se va introduce în directoriul *static*, există liniile de cod reprezentate în codul de mai jos. De asemenea, tot în *static* sunt adăugate și fișierele *offline.html* și *PushNotification.html* cu scopul de a apărea în memoria cache.

Pentru a săvârși înregistrarea la elementele de PWA, va trebui efectuată etapa de înregistrare. Browser-ul trebuie să fie încunoștințat cu locația muncitorului de serviciu (service worker). Este necesar să se indice o cale pentru scriptul în care rulează muncitorul și să se îndrume browser-ul către el. O cale din aplicația respectivă trebuie să fie asociată muncitorului de serviciu. Ulterior va trebui construit, în fișierul *JavaScript*, codul pentru muncitorul de serviciu. Din aplicația web se poate crea și o aplicație instalabilă, care să se poată descărca și pe calculator. Datele care au intrat în cache va trebui să fie fetch-uite, adică aduse pe o poziție de servire: printr-o comandă de *addEventListener* și specificându-se evenimentul.

```
1 @app.route('/service-worker.js')
2 def sw():
3     return app.send_static_file('service-worker.js')
```

Totuși, pentru a activa notificările care provin de la aplicația progresivă web, va trebuie să se realizeze o cerere de permisiune, utilizând un buton care să activeze și permită notificările de acest tip. Butonul trebuie ascuns dacă browser-ul nu acceptă notificări. După ce butonul de trimitere a mesajulurilor este apăsat, se va lansa notificarea. De asemenea, trebuie apelată și funcția *notificationButtonUpdate* atunci când este lansată aplicația, pentru a dezactiva temporar această funcționalitate a butonului.

```
1 const pushButton = document.getElementById('push-btn');
2
3 if (!("Notification" in window)) {
4     pushButton.hidden;
5   }
6 pushButton.addEventListener('click', askPermission);
7
```

```
function askPermission(evt) {
  pushButton.disabled = true;
  Notification.requestPermission().then(function(permission) {
    notificationButtonUpdate();
   });
}

function notificationButtonUpdate() {
  if (Notification.permission == 'granted') {
    pushButton.disabled = true;
  } else {
    pushButton.disabled = false;
  }
}

self.addEventListener('push', function(event) {
  console.log('[Service Worker] Push Received.');
  const title = 'Notificari';
  const options = {
    body: event.data.text(),
    icon: 'static/assets/img/1.jpg',
    vibrate: [50, 50, 50],
    sound: 'static/audio/notification-sound.mp3'
  };
  event.waitUntil(self.registration.showNotification(title, options))
   ;
});

var swRegistration = null;
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker
    .register('./service-worker.js')
    .then(function(registration) {
        console.log('Service Worker Registered!');
        swRegistration = registration;
        return registration;
    })
    .catch(function(err) {
        console.error('Unable to register service worker.', err);
    });
}
```

La partea de logare, în fișierul HTML, principalii pași ce trebuie efectuați sunt crearea labelelor în care se adaugă stilul dorit ca și atribut, după care urmează specificarea textului care descrie labela. Prin intermediul tehnologiei *Jinja2* se înregistrează datele în forma respectivă.

```
<label style="font-size: 20px;color: white">Username: </label>
<label style="margin-left: 10px; font-size: 20px">
    {{ form.username(placeholder="Username") }}
</label>
<br>
<label style="font-size: 20px; color: white">Password: </label>
<label style="margin-left: 10px; font-size: 20px">
    <div class="form-group" id="txtNewPassword2">
        {{ form.password(placeholder="Password") }}
    </div>
</label>
```

Chatbot-ul a devenit unul erudit datorită construirii a trei fișiere de timp *PyYaml*, care conțin secțiunea cu întrebări pentru care există o multitudine de răspunsuri. Totuși, întrebarea care se pune este: Cum știe chatbot-ul pe care întrebare să o selecteze astfel încât să furnizeze utilizatorului o informație sau un răspuns la întrebare apropiate de cele solicitate? Pentru a răspunde la o astfel de întrebare, una universală, s-a construit un cod, algoritmic, care îi permite chatbot-ului să dea un răspuns cât mai apropiat la întrebarea pusă; deci, chatbot-ul devine inteligent. Se citește din fișierul *.yml* scris în limba engleză (pe exemplul selectat) și se manipulează datele în funcția *convorbireXAI*, care are ca și parametru mesajul introdus de utilizator. Anume, chatbot-ul găsește cuvantul introdus de utilizator, sau rădăcina acelui cuvânt, sau parcurge propoziție utilizatorului pe cuvinte, selectând ulterior bucțile acelui cuvânt. Selectează prima apariție a celor găsite în setul de date, în baza de cunoștințe, și oferă un răspuns, schimbând inevitabil de fiecare dată răspunsul deoarece posedă câteva opțiuni de selecție. Deci, selecția răspunsului din lista constrânsă a răspunsurilor este una aleatoare, cum este și pentru o listă a răspunsurilor mai largă decât cea anterioară în cazul în care cuvântul introdus nu se potrivește cu nici o propoziție/întrebare ce se află în baza, nu mică, de cunoștințe.

```
import yaml
import random

from pathlib import Path


my_path = Path(__file__).resolve()
config_path = my_path.parent/'xai.yml'
with config_path.open() as config_file:
    config = yaml.safe_load(config_file)

def convorbireXAI(mesaj):
    l=[]
    with config_path.open() as config_file:
        xai_list = yaml.safe_load(config_file)
        for i in xai_list['conversations']:
            if i[0].lower().find(mesaj.lower()) != -1 or mesaj.lower
   ().find(i[0].lower()) == 0:
                l.append(i[1])
    if l == []:
        return xai_list['conversations'][random.randint(0, len(
   xai_list['conversations']))][1]
    else:
        return random.choice(l)
```

Parte din HTML care descrie discuția erudită și inteligentă cu XAIBOT, după ce utilizatorul a selectat tranzacțional să discute cu botul despre XAI și rațiunea botului implementat. Butoanele pentru selecția fluxului reprezintă o ramură a arborelui de decizii care există provizional din cauza setului de butoane și alegeri existente pentru a discuta cu chatbot-ul realizat.

```
    function discutie_noua() {
            var datatimp = new Date();
            var variab = '<p class="dateText"><span>' + datatimp + '
   </span></p>';
            $("#chatbox").append(variab);
            var rawText = $("#textInput").val();
```

```
            var userHtml = '<p class="userText"><span>' + '{{ current_user.name }} {{ current_user.lastn }}: ' + rawText + '</span><img src="static/media/user.ico" width="auto" height="30"></p>';
            $("#textInput").val("");
            $("#chatbox").append(userHtml);
            $.get("/get", {msg: rawText}).done(function (data) {
                var botHtml = '<p class="botText"><img src="static/media/chatbot.ico" width="auto" height="30"><span>' + 'XAIBOT: ' + data + '</span></p>';
                $("#chatbox").append(botHtml);
            });
            if(rawText.toLowerCase().localeCompare("exit")==0){
                var div1 = document.getElementById("userInput6");
                div1.style.display = "block";
                  if (div1.style.display !== "block") {
                      div1.style.display = "block";
                  }
                  else {
                      div1.style.display = "none";
                  }

                var div1 = document.getElementById("textInput_id");
                div1.style.display = "block";
                  if (div1.style.display !== "none") {
                      div1.style.display = "none";
                  }
                  else {
                      div1.style.display = "block";
                  }
            }
    }
```

În următoarea secvență de cod, extrasă din fișierul *manipulareYaml.py*, se poate observa că a fost s-a preluat textul introdus de utilizator, din adresa web, iar în lista de răspunsuri pentru această categorie se verifică dacă există și link-uri sau nu, Dacă există link-uri, se creează hyperlink-uri.

```
def findvrc(vrc):
    f = vrc.find("http")
    if f == -1:
        return vrc
    else:
        vrc = vrc.replace("(", "<a href=\'")
        vrc = vrc.replace(")", "\' target='_blank'>here<//a>")
        #print(vrc)
        return vrc
```

Fișierele *functii.py* și *chatterREF.py* sunt esențiale pentru chatbot, în toate cele trei limbi. Partea de baza în discuția cu chatbot o reprezintă partea din scriptul *functiiRO*, *functiiSRB* și *functii*.

```
from chatbot import chatterREF

def nume(text):
    discutie = 'Nice to meet you, ' + text + '. I will present you my skills. What do you want to talk about?'
    return discutie

```

```
def convorbireIT(userText):
    if userText.lower() == "yes":
        raspuns = "So, let's begin! Please type a word, syntagm or sentence, but in english. For terminate this discussion, type exit"
    elif userText.lower() == "no":
        raspuns = "I understand, I'll wait! Type YES when you are ready. In contrast, the conversation will be over."
    else:
        raspuns = "Sorry, but I don't understand! Please type YES or NO. " \
                  "I will wait for your response, but I expect you to accept. In contrast, discussion will be over."
    return raspuns

def convorbireREF(userText):
    if userText.lower() == "yes":
        raspuns = "Let's start then. Please introduce a word which you consider is necessary for XAI. I'll try to find the sentence and its associated reference. For exit, type Q"
    elif userText.lower() == "no":
        raspuns = "I understand, I'll wait! Type YES when you are ready. In contrast, the conversation will be over."
    else:
        raspuns = "Sorry, but I don't understand! Please type YES or NO. " \
                  "I will wait for your response, but I expect you to accept. In contrast, discussion will be over."
    return raspuns
```

În fișierul *chatterREF* există o funcție cheie creată de programator. Funcția *response()* permite chatbot-ului să ofere un răspun corect și un răspuns adecvat, furnizând și un link către sursa de unde a fost extras textul. Mai întâi se inițializează un obiect de tip `TfidfVectorizer` căruia i se specifică ca și atribut un tokenizer, care mai departe tokenizează vectorul de cuvinte. Utilizând algoritmul definit în funcția predefinită *cosine_similarity*, Dacă lista denumită *extragere* este vidă, sau *gigr* (variabila care indică dacă utilizatorul l-a salutat pe chatbot) nu există, adică este setată pe *None*, atunci chatbot zice că nu a înțeles ceea ce voia să-i transmită utilizatorul.

Secțiunea de cod din fișierul *views.py* care reprezintă procesul de extragere a datelor cum sunt data exactă și timpul exact când un mesaj a fost trimis, textul mesajului și numele utilizatorului, date care vor apărea pe interfața web, este reprezentată mai jos. Numele utilizatorului se extrage utilizând cunoștința căci utilizatorul conectat la aplicație este utilizatorul curent, iar textul se va transmite la fel cum se transmite și la partea de comunicare cu chatbot-ul, prin preluarea celor introduse în forme.

```
@app.route('/message', methods=['POST'])
def postMessage():
    if g.user:
        if not request.json or not 'username' in request.json or not 'messageText' in request.json:
            abort(400)
        messageData = request.json
        username = messageData['username']
        messageText = messageData['messageText']
        timeStamp = datetime.now()
        sender = User.query.filter_by(username=username).first()
```

```
        if sender is not None:
            message = Message(messageTxt=messageText, dateTime=
    timeStamp, sender_id=sender.id)
        msg = {}
        msg['timeStamp'] = timeStamp.strftime('%Y-%m-%d %H:%M:%S')
        msg['user'] = username
        msg['txt'] = messageText
        db.session.add(message)
        db.session.commit()
        socketio.emit('json_msg_response', msg, broadcast=True)
        return json.dumps(request.json)
    return redirect(url_for('login'))
```

Procesul de trimitere a fișierelor presupune specificarea unei rute individuale asociate funcției, în care se preiau numele și conținutul fișierelor, se transferă către pagina de HTML și se apelează metoda *save()*. asupra obiectului de tip *Message* cu atributele asociate. Trimiterea fișierelor este una similară cu trimiterea mesajelor. Diferența constă în numele coloanelor din tabelă (tabela *files* are coloana de conținut, *data_fc*) și în liniile de cod din fișierul *HTML*.

```
@app.route('/filesent.html', methods=["POST"])
def upload_file():
    file = request.files["inputFile"]
    fisiere = FileContinut.query.all()
    name_fc = file.filename
    data_fc = file.read()
    newFile = FileContinut(name_fc, data_fc)

    newFile.save()
    username_mes = request.form.get('username_mes', '', type=str)
    print(username_mes)
    text_mes = request.form.get('text_mes', '', type=str)
    print(text_mes)
    dateTime = datetime.now()
    mesajfile = Message(current_user.username, file.filename,
    dateTime)
    mesajfile.save()
    return render_template('layouts/default.html',
                           content=render_template('pages/filesent.
    html', newFile=newFile, fisiere=fisiere, data_fc=data_fc,name_fc=
    name_fc))
```

În aceste linii de cod se pot observa afișările pe interfața web a mesajelor, respectiv a fișierelor. Totul constă în valoarea parametrului filei, prin intermediul căruia se detectează dacă mesajul introdus este un fișier sau nu.

```
<div id="messageList">
    <div class="cls3">
        {% for m in messages %}
            <ol id="demo">
            {% set filei=0 %}
            {% for fc in fisierele %}
                {% set filei=1 %}
                {% if m.text_mes == fc.name_fc %}
                    {% set filei=1 %}
                    {% if filei == 1 %}
                        <div class="messageFlex">
                          <div >
```

```
                                <div >
                                    <strong style="margin-left: 12px;
    color: #7b38d8">{{ m.username_mes | safe }}</strong>: <a href="{{
    url_for('download_file') }}"> {{ m.text_mes }}</a>
                                </div>
                                <div class="timeStamp" id="timeStamp"><i
    >{{ m.dateTime }}</i></div>

                            </div>
                        </div>
                    {% endif %}
                {% endif %}
            {% endfor %}
            {% if filei == 0 %}
                {% for fc in fisierele %}
                    {% if m.text_mes != fc.name_fc %}
                    <div class="messageFlex">
                        <div>
                                <div >
                                    <strong style="margin-left: 12px;
    color: #7b38d8">{{ m.username_mes | safe }}</strong>: {{ m.
    text_mes }}
                                </div>
                                <div class="timeStamp" id="timeStamp"
    ><i>{{ m.dateTime }}</i>
                                </div>
                        </div>
                    </div>
                    {% endif %}
                {% endfor %}
            {% endif %}
            </ol>
        {% endfor %}
    </div>
</div>
```

In fișierul *default.html* din cadrul directoriului *layouts* se importează toate librăriile necesare pentru reprezentarea grafică, incluzând partea de Progressive Web Application, partea de utilizare de *bootstrap* și partea de import a librăriilor HTML necesare pentru navigare și informare, cum sunt *navigation.html*, *sidebar.html* și *footer.html*.

```
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?
    family=Montserrat:400,700,200" />
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font
    -awesome/latest/css/font-awesome.min.css" />
    <link href="/static/assets/css/bootstrap.min.css" rel="stylesheet
    " />
    <link href="/static/assets/css/light-bootstrap-dashboard.css?v
    =2.0.0 " rel="stylesheet" />
    <link href="/static/assets/css/demo.css" rel="stylesheet" />

    <link rel="manifest" href="{{ url_for('static', filename='
    manifest.json') }}">
    <script type="text/javascript" src="{{ url_for('static', filename
    ='javascript/install.js') }}"></script>
    <script type="text/javascript" src="{{ url_for('static', filename
    ='service-worker.js') }}"></script>
```

Aceasta este secțiunea de cod menită structurării paginilor ce reprezintă interfața grafică de tip web.

```
<div class="container-fluid" style="font-family: Roboto;">
    <div class="bar-lateral">
        {% include 'includes/sidebar.html' %}
        <div class="main-panel">
            {% include 'includes/navigation.html' %}
            {{ content | safe }}
            {% include 'includes/footer.html' %}
        </div>
    </div>
</div>
```

Această secvență de cod este destinată creării unei aplicații *responsive*, care să fie centrată și disponibilă sub o formă adecvată și lizibilă pe dispozitivele Android.

```
<meta content='width=device-width, initial-scale=1.0, maximum-scale
   =1.0, user-scalable=0, shrink-to-fit=no' name='viewport' />
```

Au fost descrise, reprezentate, analizate și dezvoltate părți ale implementării, precum și idei de implementare. Cele specificate reprezintă unele bucăti esențiale în crearea, dezvoltarea, realizarea, implementarea și testarea aplicației software, cu tendința de a construi și a obține o aplicație funcțională, complexă, interesantă, și folositoare societății. Întreg codul aplicației poate fi găsit pe pagina de *GitHub*, asociată repozitoriului în care au fost introduse fișierele.

Capitolul 5

Arhitectura aplicației

Arhitectura aplicației constă în oferirea unei analize și descrieri a aplicației, incluzând toate șabloanele de proiectare, precum și tehnicile de implementare a datelor sau a bordului de lucru. Cu această ocazie este oferită o diagramă mai amplă pentru descrierea funcționalităților și a modului de structurare a datelor și informațiilor. Toată filosofia proiectării și realizării proiectului începe de la secvența controller-ului unde sunt specificate fișierele esențiale de pornire și rulare a unei aplicații de tip Flask în Python. S-a configurat baza de date, cu precizările că este vorba de o bază de date relațională SQLite, indicându-se ruta unde se crează baza de date după prima rulare a aplicației. Următorul fișier important este cel de construire a modelelor pentru baza de date, care presupune stabilirea conectivității cu librăria $flask\_sqlalchemy$ și clasa principală $SQLAlchemy$. Se va contrui un obiect de tipul clasei principale. Ulterior se implementează clasele necesare pentru construirea tabelelor în baza de date, precizându-se și metodele claselor. Exemple de metode predefinite utilizate în cod sunt: *add()*, *commit()*, *rollback()* și *delete()*. În fișierul *configuration.py* s-a realizat configurarea bazei de date prin setarea a câtiva parametri și inițializarea căii de creare și stocare a bazei de date. Pentru a îndeplini o conectivitate între partea de back-end și cea de front-end, când vine vorba de stocarea și salvarea datelor în bază, atunci este necesară crearea formelor, în fișierul $forms.py$. Fișierul cu vederi este esențial în crearea rutelor prin instanțierea unui obiect de tip Flask și prin accesarea ulterioară a metodei $route()$.

În fișierul cu vederi, denumit *views.py*, s-a efectuat prin intermediul tehnologiei *Jinja2* conexiunea cu paginile HTML, printre care cele mai importante sunt, la primul acces al aplicației, partea de logare și înregistrare. Pentru anumite cazuri era necesar să se realizeze o legătură între utilizatorul curent și baza de date. Aceasta s-a efectuat prin intermediul valorilor specificate cu $Jinja2$, *current_user.username* și *current_user.password*.

Elementele de aplicație web progresivă (PWA), care s-au utilizat la dezvoltarea acestei aplicații, sunt notificarea de tip *push*, în care toți utilizatorii logați în aplicație sunt notificați după ce este postat un mesaj pe grup, apoi *geolocation*, care-i permite utilizatorului să-și activeze locația; al treilea element este *Background Sync*, care permite funcționarea aplicației și într-un mod off-line. Elementele de aplicație web progresivă s-au construit în cadrul fișierelor de JavaScript cu numele *app.js* și *service-worker.js*. Toate elementele de PWA sunt corelate cu crearea platformei de mesagerie. Platforma denumită *intellBot TK* este destinată utilizatorilor, se presupune, eruditi, care comunică între ei și se pregătesc pentru diferite quiz-uri, iar conversația

se desfășoară prin intermediul prizelor web, importând clasa *SocketIO* din librăria **flask_socketio**. Chatbot-ul trilingv inteligent și conversațional le stă utilizatorilor la dispoziție pentru ajutor.

Partea de construire și dezvoltare a chatbot-ului este constituită din câteva fișiere Python și HTML. Pentru ca cele mai importante librării din Python/Flask să nu se instaleze fiecare pe rând introducând comanda *pip3 install* ***nume_librărie***, ci în mod automat, au fost introduse în fișierul text *requirements.txt* și *nltk.txt*.

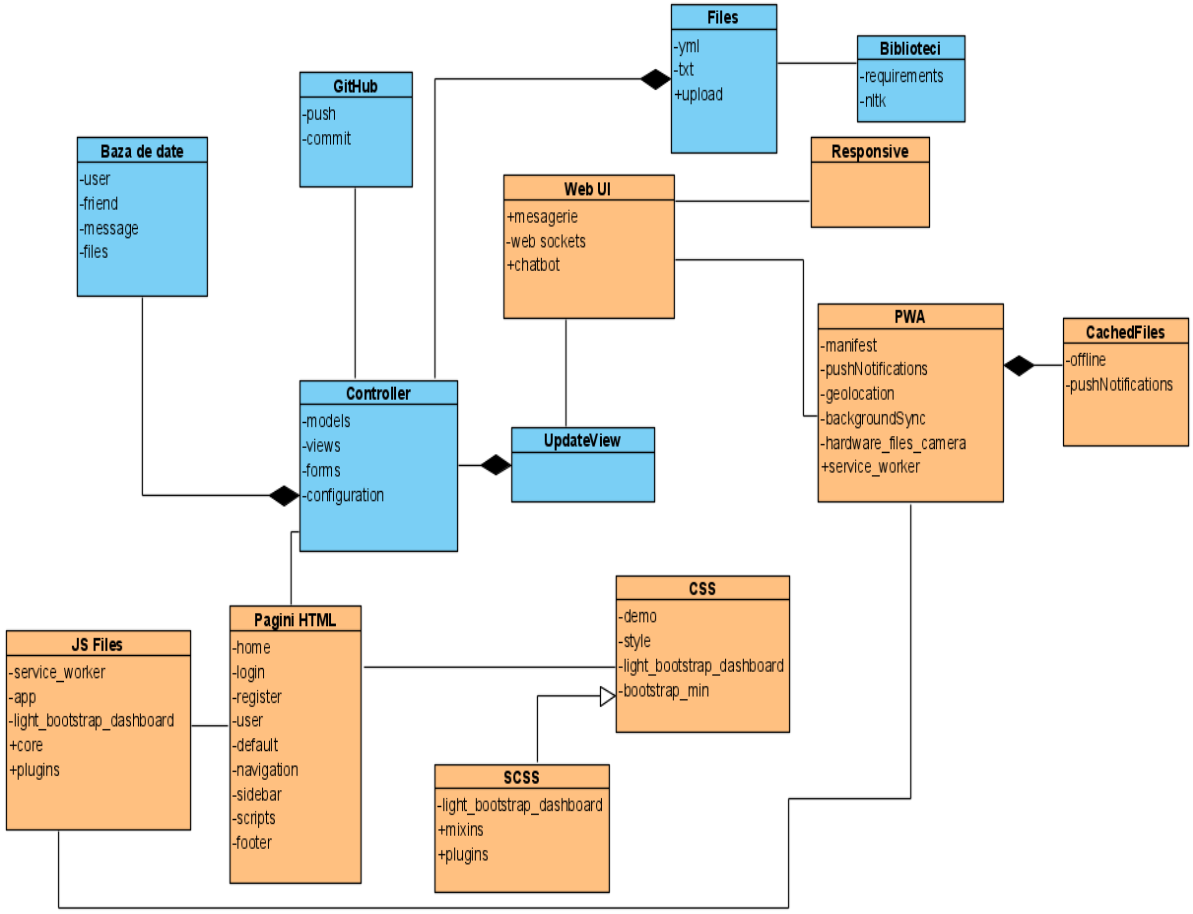

Figura 5.1: Arhitectura la nivel de macro-plan

Diagrama de clase va specifica arhitectura sistemului expert în care observăm o arhitectură de tipul *Model-View-Controller*. Această diagramă este destinată descrierii aplicației la nivel de macro-plan arhitectura, cu precizarea avantajelor și a dezavantajelor de care dispune această personalitate virtuală. Pentru construirea chatbot-ului s-au folosit diferite comenzi din bibliotecile *Scikit-learn* și *Natural Language Toolkit*.

Biblioteca *chatterbot* din Python generează răspunsuri pentru textul introdus de utilizator. Folosește mai mulți algoritmi din domeniul învățării automate pentru a produce răspunsuri pentru mai multe sectoare de discuție. Aceste răspunsuri sunt mai precise decât dacă se utilizează metode definite manual. Chatbot-ul se proiectează folosind librăria *chatterbot* astfel încât să fie instruit în trei limbi diferite pentru domenii multiple. Un obiect de tip *ChatBot* permite oferirea unui răspuns îmbunătățit la mesajul trimis de utilizator. Pentru un text de input, pe care îl obține chatbot-ul de la utilizator, chatbot-ul are un set de răspunsuri (din baza de cunoștințe) pe care

le poate furniza. Cu aceasta, acuratețea chatbot-ului crește. Programul selectează cel mai apropiat răspuns de potrivire din cea mai apropiată instrucțiune de potrivire care se potrivește cu textul de intrare, iar ulterior alege propoziția din secțiunea cunoscută de instrucțiuni. Pentru partea de antrenare se acordă chatbot-ului o listă de acțiuni cu care acesta este instruit.

Pagina în care se discută cu chatbot-ul se actualizează după fiecare mesaj, fie al utilizatorului, fie al chatbot-ului. Paginile de HTML au fost completate cu diferite fișiere *CSS* și *Bootstrap* destinate proiectării aplicației. Partea în care este creat și dezvoltat doar chatbot-ul trilingv conversațional inteligent este urcată pe Heroku prin intermediul diferitelor comenzi *Git*.

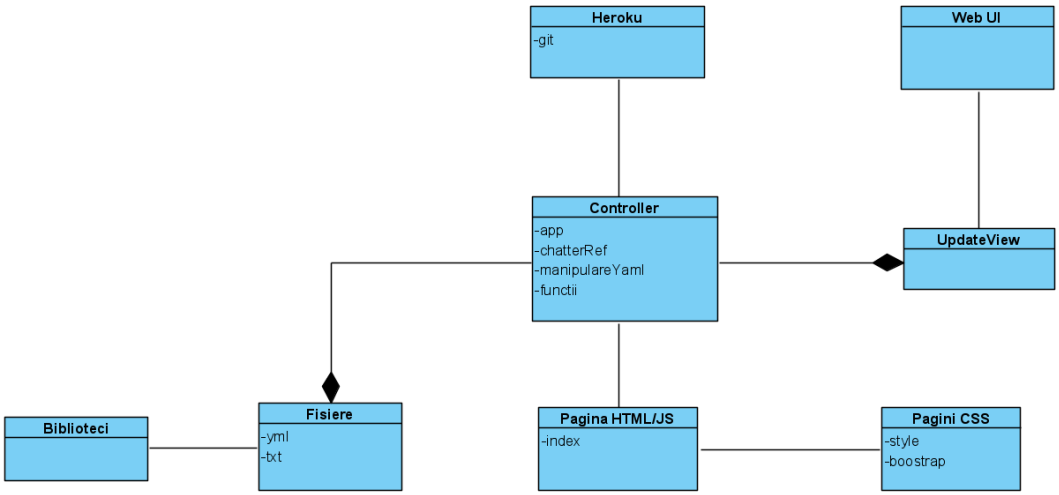

Figura 5.2: Arhitectura sistemului inteligent

Baza de date este creată prin intermediul fișierului *configuration.py*, prin intermediul modelelor și al obiectului de tip *SQLAlchemy* din clasa *flask_sqlalchemy*. Tabelul *files* conține numele fișierului și conținutul acestuia prin intermediul coloanei care păstrează date de tip *BLOB*. Tabelul *friend* era destinat pentru doi utilizatori conectați între ei și introduși în lista de prieteni. Totuși, această idee va fi implementată pe viitor. Tabelul *message* conține datele principale despre mesaj, mai precis numele utilizatorului, conținutul, data și timpul când acesta a fost trimis. Pe de altă parte, tabelul *user* conține toate datele personale ale utilizatorului introduse în baza de date la partea de înregistrare în aplicație.

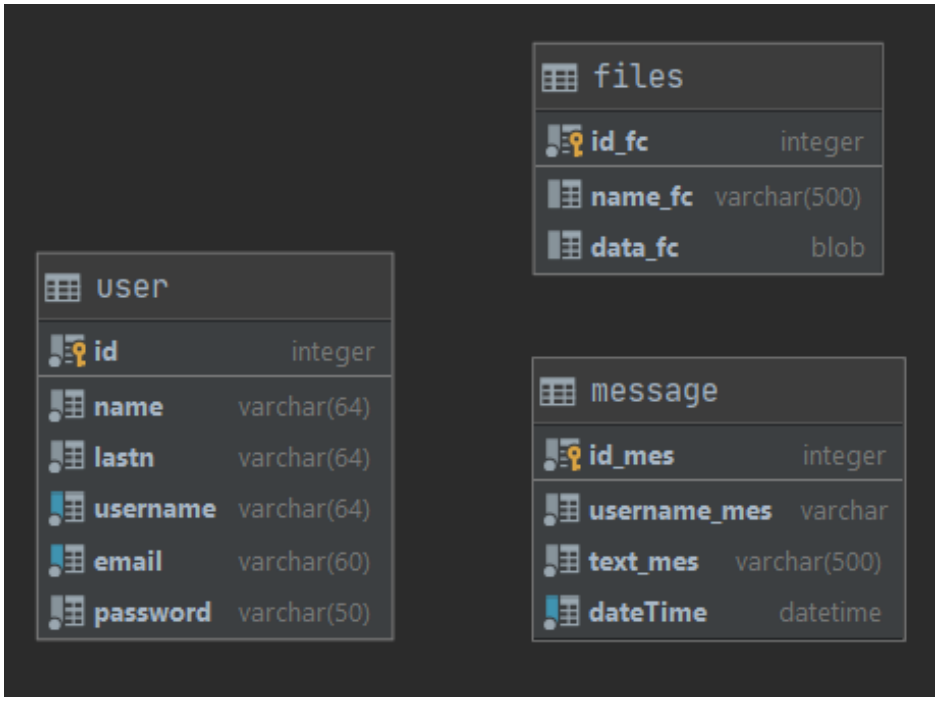

Figura 5.3: Diagrama entitate-relație

Capitolul 6

Concluzii și direcții viitoare

Utilizarea tehnologiei Flask pentru a crea rute, care ajută la conectarea codului din Python cu cel din HTML, după caz și prin intermediul tehnologiei Jinja, au fost unele dintre lucruri pe care le-am utilizat deja în proiectul efectuat pentru firma Nokia, la materia *Proiect Colectiv*, însă pot să recunosc că am și progresat de atunci. Anume, am reușit să conectez direct codul din Python cu cel din JavaScript prin *JQuery* și metoda *get*. La lucrarea de licență am dobândit cunoștințe din domeniul de învățare automată și procesarea textului în limbaj natural. Am descoperit utilitatea bibliotecilor *Natural Language Toolkit* și *Scikit-Learn*. Am învățat o nouă modalitate de a face *split* asupra propozițiilor dintr-un fișier text; prin metoda *sent_tokenize(text)* din biblioteca *nltk*. De asemenea, am descoperit fișierul *Yaml* pentru stocarea datelor sub o formă structurată, precum și citirea din fișierul respectiv. S-a obținut un chatbot conversațional, care poate discuta cu utilizatorul despre aproape orice domeniu. Utilizatorul întreabă, iar chatbot-ul răspunde, sau vice versa în cazul în care asistentul virtual nu are capabilități de a oferi un răspuns nici măcar apropiat. Partea cu chatbot-ul de la aplicație se găsește la adresa `https://testare-herokuflask.herokuapp.com/`. Deci, a fost pusă pe Heroku utilizând comenzi în *Command Prompt*. Această abilitate a fost dobândită iarăși pe parcursul efectuării lucrării de licență pentru prima oară. Până acum, am reușit să urc aplicații doar pe *PythonAnywhere*.

O parte din această aplicație, mai precis cea cu dezvoltarea și comunicarea cu chatbot-ul erudit și inteligent, este publicată ca și aplicație separată de Universitatea de Vest, la adresa: `https://chatbot.ai.uvt.ro` iar ca și o componentă apare pe site-ul XAION: `https://xaion.uvt.ro`.

În ceea ce privește direcțiile viitoare, poate fi introdusă partea de împrietenire cu alți utilizatori, care ulterior va permite procesarea părții de construire a conversației directe, private. De asemenea, tabela *friend* cu coloanele, caracteristicile și constrângerile proprii, va deveni una relevantă în aplicație, fapt care va fi valabil și altor secvențe de cod și chiar funcții sau metode ale claselor (de pildă, clasa din *models.py* sau *forms.py*). Partea de construire a ambientului în care pot comunica doar două persoane va necesita crearea unei camere, care se va distinge de alte pagini prin adăugarea căii, construite din numele de utilizator ai ambilor utilizatori, la finalul adresei web și rutei deja create pentru conversație și platforma *intellBot TK*. Pentru sectorul în care comunică între ei utilizatorii eruditi va fi necesar să se adauge roluri pentru utilizatori. Mai precis, administratorul să fie acela care să șteargă mesajele sau fișierele (în cazul în care le consideră inadecvate sau nedemne de a exista pe platform *intellBot TK*) trimise de alți utilizatori.

Un lucru adițional care poate fi adăugat la aplicație îi va oferi utilizatorului posibilitatea de a transmite mesaje vocale[Raj19] lui XAIBOT. Aplicația poate fi integrată cu diferite soluții software și API specializate. Se poate folosi ca și extensie la Google Browser astfel încât să fie un suport utilizatorului.

Eventual s-ar putea adăuga un rol de admin asociat utilizatorului principal, administrator al aplicației/platformei. Acesta va avea privilegiul de a șterge fișierele deja completate cu răspunsuri și mesajele neadecvate în conversația erudită care se desfășoară pe platforma *intellBot TK*. Un lucru adițional ar fi și posibilitatea de resetare a parolei în cazul în care utilizatorul a uitat-o.

La fel, se va plasa aplicația finală pe site pentru construirea aplicației mobile prin intermediul tehnologiei *WebView*, iar codul final integral va fi postat pe *GitHub*.

Bibliografie

[ADRS+20] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. 58, June 2020.

[BKL19] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python.* O'Reilly, first edition, 2019.

[Joh19] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib.* Apress Media, second edition, 2019.

[KZH17] Amit Kothari, Rania Zyane, and Joshua Hoover. *Chatbots for eCommerce: Learn How To Build a Virtual Shopping Assistant.* Bleeding Edge Press, 2017.

[NJG18] Gayatri Nair, Soumya Johnson, and V. Sathya (Guide). Chatbot as a Personal Assistant. *International Journal of Applied Engineering Research*, 13, 2018.

[Raj19] Sumit Raj. *Building Chatbots with Python.* Apress Media, first edition, 2019.

[Zor19] Jaime Zornoza. Deep Learning for NLP: Creating a Chatbot with Keras. *KD nuggets*, 2019.