Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

MIIKKA YLI-HONGISTO

Chattibotin sovellus laajaan käyttöön

TIETOJENKÄSITTELYN KOULUTUSOHJELMA
2020

Tekijä(t)	Julkaisun laji	Päivämäärä
Yli-Hongisto, Miikka	Opinnäytetyö, AMK	Syyskuu 2020
	Sivumäärä	Julkaisun kieli
	64	Suomi

Julkaisun nimi

Chattibotin sovellus laajaan käyttöön

Tutkinto-ohjelma

Tietojenkäsittely

Chattibotit ovat ajan myötä nousseet esille hyvänä vaihtoehtona rutiinitoimenpiteiden automatisoinnille. Bottien toiminta perustuu ohjelmoituihin reaktioihin, joita käyttämällä botti pystyy käymään asiakkaan kanssa läpi keskustelun, mikä vähentää tarvittavien päivystäjien määrää yksittäisen botin pystyessä hoitamaan monta henkilöä samaan aikaan.

Projektin tarkoituksena oli käydä läpi botin rakentamista käyttämättä valmispohjia, ja samalla tuottaa esimerkki Foredata-yhtiölle siitä, miten työläs itserakennettu chattibotti tulisi olemaan ja kuinka paljon hyötyä heille sellaisesta olisi. Suunnitelmaan kuului rakentaa Python-kielellä chattibotti, joka pystyi vastaamaan kysymyksiin, keräämään asiakkaan nimen ja työtaidot, ja tallentamaan ne verkkoon myöhempää käyttöä varten. Työssä käytiin myös yleisesti läpi bottien historiaa, rakennusfilosofiaa ja suosituksia.

Botin rakennukseen käytettiin pääasiassa Rasa.comista sovellettua esimerkkiä, jossa asiakkaiden oletetuista aikomuksista (intents) rakennettiin niihin perustuen botille vastaukset. Jotkin vastauksista tarvitsi priorisoida erilleen muista, joka johti actions.py tiedoston rakentamiseen. Kyseiseen tiedostoon rakennettiin myös koodi, joka keräsi tiedot aluksi niille varatuille paikoille (required slots), ja sitten tallensi ne google driveen.

Tuloksena syntynyt botti hyväksyttiin Foredatalla hyväksi esimerkiksi, mutta tulokseen virallisen käyttöönotettavan chattibotin tuottamisesta ei vielä päästy.

Asiasanat

Ohjelmointi, Ohjelmistotuotanto, Ohjelmistosuunnittelu, asiakaspalvelu, Tietojenkäsittely

Author(s)	Type of Publication	Date
Yli-Hongisto, Miikka	Bachelor's thesis	September 2020
	Number of pages	Language of publication:
	64	Finnish

Title of publication

Applying chatbots for wide usage

Degree program

Data processing

Chatbots have over time risen as a good option for automating some tasks. The bots functions are based around preprogrammed reactions, which the bot can use to have a conversation with clients. This lowers the amount of personnel needed when a single bot can handle multiple clients at the same time.

The purpose of this project was to go through the building process of a chatbot, without using any readied templates found on the web, as well as producing an example for a company named Foredata about how work intensive a self-produced bot would be and how much use it would be for them. The plan included building a python-based bot, which was able to answer questions, collect the clients name and job skills, and then save them onto cloud services for later use. Thesis is also explained the history, building philosophy and recommendations for the bot.

Primary example used for building the bot was adapted from the examples provided at Rasa.com, which used the intents provided by the clients and build the responses of the bot around them. Some answers were prioritized, leading into the creation of actions.py program. Same program also included the code to gather the info for the required slots, and then proceeded to save them onto google drive.

The chatbot produced was accepted as a good example by Foredata. However, they haven't yet come to a result about producing a specialized bot for their company.

Key words

Customer Service, Programming, Software design, Data Processing

SISÄLLYS

LYHENTEET JA TERMINOLOGIA

BOTTI	Botti on ohjelmisto, joka on ohjelmoitu toimimaan itsenäisesti jollain tapaa. Referoin chattibottiin opinnäytetyön aikana usein pelkkänä bottina.
CHATTIBOTTI	Chattibotti on chattiohjelmassa toimiva botti, joka vastaa siihen ohjelmoituihin intentteihin siihen ohjelmoiduilla vastauksilla ja toiminnoilla.
INTENT	Aikomukset. Botille ohjelmoitu lausahdus tai sana, jonka luettuaan botti vastaa sille ohjelmoidulla vastauksella.
RESPONSE	Vastaukset. Jokaiselle intentille on oma vastaus, mitä botti käyttää kun intenttiin liittyvä lause tai sana sanotaan.
STORIES	Botin stories.md tiedostoon kirjoitettu metodi, joka kertoo sille oletetun järjestyksen sille ohjelmoiduille intenteille ja responseille.
LIBRARY	Python-kielen toiminta perustuu open source libraryssa oleviin kirjastoihin, joista se nostaa ohjelmistotietoja. Kutsun näitä tekstin aikana yleisesti kirjastoiksi.
SLOT	Paikka asiakkaan antamaa tietoa varten. Jos botin tarvitsee tallentaa tai muistaa asiakkaan antamaa tekstiä, se tarvitsee slotin jokaista erillistä tietoa varten.

1 JOHDANTO

Jo vuonna 2018 Suomessa oli tekoälyyn perustuvia botteja käytössä. Digibarometrin mukaan niiden käyttö oli kuitenkin vielä vähäistä. Esimerkiksi Elisa oli käyttänyt tekoälyä etsiäkseen asiakkaille erityisesti kohdennettua mainontaa ja Vaisala käytti sitä energiantuoton ennustuksiin. Kuitenkin tähän aikaan nostettiin esille, että ongelmana yleisen käytön hitaaseen nousuun oli siiloutuminen. Siiloutuminen tässä tilanteessa tarkoittaa vuorovaikutuksen hidastumista tarkkaan ohjatun toiminnan takia. Lisäksi monet suomalaisyhtiöt eivät nähneet bottia tarpeelliseksi yhtiönsä asioiden hoitamiseen. (Keränen 2018.)

Ajan kuluessa mielipide on muuttunut positiivisempaan suuntaan. Väitöskirjatutkija Sami Koivunen kirjoittaa artikkelissaan “Case-esimerkit valottivat bottien mahdollisuuksia Chatbot Day –seminaarissa” (2019) monista erilaisista kuulemistaan kokemuksista chattibottien mielekkäästä käytöstä.

Ollessani työharjoittelussa Foredatalla työnantajani otti usein esille kiinnostuksensa chattibotin käytöstä omilla verkkosivuillaan. Samana vuonna kuin Keräsen artikkeli (2018) julkaistiin, minulle oli annettu tehtäväksi tutkia, olisiko itseohjelmoidusta botista Foredatalle hyötyä. Tähän aikaan tutkin Chatfuelin avulla asiaa ja opin niiden toimintaa yksinkertaisimmalla mahdollisella tavalla. En päässyt tyydyttävään lopputulokseen työharjoitteluni aikana ja bottitilanne jäi epäselväksi. Tästä syystä kysymykseksi itselleni nousi, voisinko selvittää ongelmaa opinnäytetyössäni.

Tulen jatkamaan seminaarityötäni lopputyöksi. Lopputyöni tarkoituksena on saada selville, sopisiko chattibotti Foredata-yhtiön toimintaan, ja kuinka hyödyllinen se olisi heille. Tulen käsittelemään pääasiassa chattibotteja, mutta käsittelen myös muita botteja jonkin verran. Teoriassa käyn läpi sekä bottien toimintaperiaatteita että niiden historiaa. Suurin osa tekstistä käsittelee bottien positiivisia ja negatiivisia puolia,

koska tekstin tarkoituksena olisi kannustaa virallisen botin käyttöönotto. Käytännössä tulen käymään läpi botin rakennusfilosofiaa, ja rajaan samalla, millaiseksi aion rakentaa oman Python-pohjaisen chattibottini, jolla tulemme tätä testaamaan.

2 CHATTIBOTIT

2.1 Mikä on chattibotti?

Yksinkertaisimmillaan chattibotilla tarkoitetaan sovellusta, joka tunnistaa sille annettuja kysymyksiä tai sanoja, ja antaa niiden perusteella valmiin vastauksen. Chattibotit eivät itse opi mitään, mutta niiden aiempia keskusteluja tarkkailemalla koodiin voidaan lisätä aina uusia vastauksia tarpeen mukaan. (Hintikka 2020.) Chattibottien iso osa on myös niiden rakennusfilosofia: sen sijaan, että ne olisi rakennettu tietynlaisia asiakkaita varten, chattibotit rakennetaan kaikkien ikäluokkien käyttöön. Muihin ohjelmiin verrattaessa botti kysyy paljon enemmän kysymyksiä varmistaakseen, että se tekee kaiken oikein. Lopputuloksena täytyisi olla botti, jonka kanssa puhuminen on mahdollisimman samankaltaista kuin puhuisi ihmisen kanssa. (Raj 2019, 2.)

Boteista on monenlaista hyötyä työelämässä sekä itse asiakkaalle että sen omistajalle: ensinnäkin, ne ovat helppoja käyttää ilman aiempaa kokemusta. Botille voi alkaa esittää kysymyksiä milloin tahansa ja mistä tahansa, kunhan sinulla on jonkinlainen laite sitä varten. Botit voivat myös olla päällä 24 tuntia vuorokaudessa vaatimatta työntekijää valvomaan sen toimintaa. Hyvät botit pystyvät myös hoitamaan tuhansia asiakkaita kerralla, ja yhden botin rakennus on paljon halvempaa kuin monen työntekijän palkkaaminen. Tämän lisäksi botti voi myös muistaa asioineen asiakkaan aiemmin tekemät kysymykset ja päätökset paremmin kuin ihminen. (Raj 2019, 3, 5.)

Sami Koivunen nostaa esille myös, että tällä hetkellä chatbottien tehtävät koostuvat pääasiassa asiakaspalvelusta käyttäjän nähtävillä olevassa käyttöliittymässä. Nykyaikana chattibotit voidaan esimerkiksi jakaa hook- ja learn-botteihin. Hook-botit keräävät tietoa asiakkaan antamilla kohdesanoilla, kun taas learn-botit tarjoavat sitä vakuutuskysymyksillä. Learn-chatbotit pystyvät jopa luokittelemaan asiakkaiden kysymyksiä niiden tavoitteiden mukaan. Näiden bottien opetusprosessia varten on yrityksissä nimetty ‘bottikuiskaaja’, joiden tehtävänä onkin ohjelmoida botteja ymmärtämään monien eri lauseiden päätymisen samaan vastaukseen. Botit voidaan integroida myös erilaisiin chattiohjelmiin, kuten Slackiin ja Discordiin, automatisoiden yksitoikkoisempia toimintoja yksinkertaisiin pyyntöihin. (Koivunen 2019.)

Bottien, ja varsinkin chattibottien, hyöty on yksinkertaisten toimintojen automatisointi. Jos botille on annettu jokin tieto kerran, ne pystyvät toistamaan sen loputtomiin kaikkia tulevia tarpeita varten. Tässä mielessä ne voivat olla verrattavissa ohjelmointirajapintoihin, jotka olivat jo käytössä ennen bottien rakennusta. Perustoiminta on molemmissa sama: niille annetaan tietoa, ja sitä käyttämällä ne antavat sinulle vastauksen. Jotkut botit voivat jopa käyttää ohjelmointirajapintoja hyväkseen omia tietojaan hankkiessa tai tehtäviään tehdessä. Botteja käyttämällä kuitenkin voidaan ohittaa kokonaan varsinaisella verkkosivulla käynti, jolloin botti voi esimerkiksi varata automaattisesti hotellista huoneen. (Raj 2019, 1.)

Chattibotit eivät tietenkään ole ongelmattomia. Suurin osa ongelmista on niiden tuottajilla ja ohjelmoijilla: Mahdolliset asiakkaat ovat vaikea kohde löytää ilman suurta määrää mainostukea, eivätkä ne ole vielä niin kuuluisia, että jokainen web-sivun omistaja sellaisen uskoisi tarvitsevansa. Samalla tavalla chattibottien hinnoittelu on vielä hyvin epäselvä, koska kukaan ei ole varma, minkä arvoisia ne ovat. Jotkut otaksuvat, että jossain vaiheessa niitä voitaisiin myydä samalla tavalla kuin puhelinappeja pienellä hinnalla. Loppujen lopuksi boteista tuleva hyöty on selvempi suurilla yhtiöillä, jotka haluavat helpottaa asiakaspalveluaan, kuin yksittäisille henkilöille. (Chatbot Community 2017.)

2.2 Chattibottien historiaa

Ensimmäisenä bottina voidaan pitää saksalaisen Joseph Weizenbaumin rakentamaa ohjelmaa ‘ELIZA’. Weizenbaum rakensi sen hänen kiinnostuksensa herättyä Alan Turingin tuottaman Turingin kokeen johdosta. Sen mukaan konetta, joka pystyy puhumaan ihmismäisesti niin hyvin, että saisi vakuutettua sen kanssa puhuvan henkilön ihmisyydestään, voidaan pitää älykkäänä. ‘ELIZA’ oli rakennettu tämä mielessä: se käyttäytyi kuin terapeutti, ja yritti huijata ihmiset uskomaan, että se oli oikea henkilö. (Salecha 2016.)

‘ELIZAn’ toimintaperiaate perustui yksittäisten sanojen ymmärtämiseen ja hyvin laajoihin kysymyksiin. Esimerkiksi, jos sille kerrottiin “äitini tekee hyvää ruokaa”, botti tunnistaisi sanan äiti, ja vastaisi aina kysymyksellä, esimerkiksi “minkälainen

perheesi on?” Tämä toistuva toiminta sai botin vaikuttamaan hyvin ymmärtäväiseltä, vaikka se vaan seurasikin koodattuja komentojaan. (Salecha 2016.)

ELIZAn rakentaminen inspiroi muita botteja, kuten esimerkiksi vuonna 1971 samalla tekniikalla rakennettu Parry, joka oltiin rakennettu käyttäytymään niin kuin vainoharhainen potilas. Parryssä oli kuitenkin selvänä erona sen toiminta. Sen sijaan että se yrittäisi vaikuttaa ymmärrettävältä, se pystyi myöntämään välinpitämättömyytensä, vaihtamaan keskustelun aihetta, ja jopa lisäämään keskusteluun mafiaepäilyjä. Parryn avulla tultiin siihen tulokseen, että ihmisten tunnepuolta on helpompi imitoida kuin älyllistä, mutta toiminnallisesti se oli silti vain illuusio. (Pereira 2018.)

Vuonna 2003 Juergen Pirner voitti Loebner-palkinnon Jabberwacky-nimisellä botillaan. Hänen mielestään chattibottien mahdolliset virheet olivat aina sen ohjelmoijan eivätkä varsinaisen ohjelman syytä. Maria Pereiran Essee Chatbots Greetings to Human-Computer Communication jopa vertaa ohjelmoijan, botin tekijän roolia kirjailijaksi. Samalla tavalla monelle botille annetaan tapa, jolla käyttäytyä, jolloin niille syntyy persoonallisuus. (Pereira 2018.)

Jotkin botit, kuten esimerkiksi Jabberwacky ja Cleverbot, oppivat uusia vastauksia tallentamalla saamiaan tuntemattomia vastauksia, ja käyttämällä niitä muissa tilanteissa. Samalla tavalla, Fred, ”Functional response emulation device”, rakensi omia lauseitaan keräämistään tiedoista. Alussa sille oli ohjelmoitu vain peruskomennot, joilla se pystyi käymään keskusteluita sitä opettavien tai satunnaisten ihmisten kanssa. Nykyaikana tällaiset oppitavat voivat olla riskialttiita, kun verkon käyttäjien määrä kasvaa, mutta ne antavat hyvän kuvan, mihin botit voivat pystyä. (Pereira 2018.)

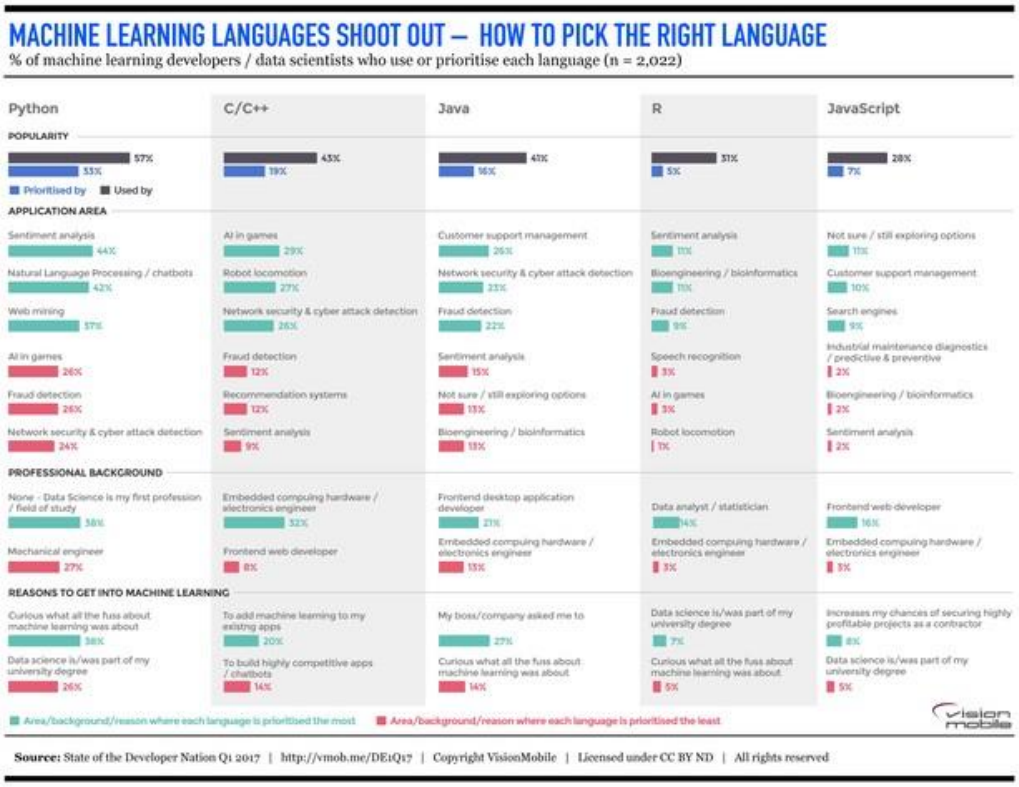
Esimerkkinä suomalaisesta chattibotista on vuonna 2019 YLE-kioskin, Sitran ja Suomen Akatemian rakentama ’Vaalibotti’. Sitä kehuttiin paljon ensimmäisenä chattibottiteknologiaa käyttävänä vaalikoneena, jonka tarkoituksena oli auttaa varsinkin nuoria äänestysikäisiä halutun ehdokkaan löytämisessä. Tarkoituksena hankkeella oli saada vaikutushaluiset nuoret äänestämään helpottamalla ehdokkaan valintaprosessia. (Nissinen 2019.) Botin toimintaperiaate on hyvin yksinkertainen: se tarjoaa erilaisia aiheita, jotka on rakennettu YLEn uutiskysymyksistä, ja selittää niistä käyttäjälle hieman. Sitten Chattibotti antaa botin käyttäjälle kysymyksen aiheesta, ja 5

vaihtoehtoa, mistä valita vastaukseksi yhden: Täysin samaa mieltä, vähän samaa mieltä, vähän eri mieltä, täysin eri mieltä ja kysymyksen ohittamisen. Tämän jälkeen botti ehdottaa vastausten perusteella käyttäjälle ehdokkaan. Kun botilla on tällainen rakenne, sen kanssa ei käydä varsinaista keskustelua, mutta sillä saadaan kerättyä selvemmin tietoja, mitä tarvitaan. Tämä saattaa toimia hyvänä esimerkkinä oman bottini toiminnasta tulevaisuudessa.

2.3 Ohjelmointikielet

Nykyaikana bottien rakennus on hyvin helppoa, ja vaihtoehtoja niiden rakennuskieleen on monia. Vanhimpana voidaan pitää XML-pohjaista AIML:ää, Artificial intellicence markup languagea, joka rakennettiin 1990-vuosikymmenen loppupuolella. Koska se on osa XML:ä, sen toiminta perustui ’tagaamiseen’ eli botti reagoi tiettyihin sanoihin ja pystyi antamaan erilaisia vastauksia koodin perusteella. Ongelmaksi systeemillä nousee kuitenkin se, että yksinkertaista keskustelua varten tarvitsee kirjoittaa paljon koodia, jolloin sen kirjoituksesta tulee hyvin työlästä. (AIML foundation 2018.)

Myös yleisillä ohjelmointikielillä voidaan ohjelmoida botti. Esimerkiksi Joe Lobo vertaa pythonia linkkuveitseen sen oppimishelppouden takia. Lisäksi, kuvassa 1. vuoden 2017 mielipidekyselyssä tultiin tulokseen, että kahdesta tuhannesta datatieteilijästä ja koneoppimiseen erikoistuneista suunnittelijasta 57% käytti sitä, ja 33% priorisoi sen käytön muihin kieliin verrattuna. Chattibottien ohjelmointia varten Pythonilla on hyvin paljon avointa lähdekoodia yleiseen käyttöön, mikä teki monien toimintojen rakennuksen helpoksi. (Lobo & Fontseca 2017.) Itse valitsin kyseisen kielen omaa työtäni varten. Se on ollut kielenä hyvin mukava kirjoittaa, ja ongelmien löytäminen koodista on helpompaa kuin muissa kielissä. Ongelmaksi yleensä nousee sen tarkkuus esimerkiksi sisennyksien määrästä, jolloin pieni virhe pysäyttää koodin kokonaan, tai estää sitä tunnistamasta mitä sille on kirjoitettu. (Kuva 1.)



Kuva 1: Mielipidekyselyn tulokset oikeasta kielenvalinnasta koneoppimiseen, Christina Voskaglau, ’what is best language for machine learning?’

Kielenä bottien kirjoittamiseen Pythonilla on heikkoutena sen hitaus ja epäselvyys lukiessa. Koodin toteutusvaiheessa koodin tulokset tulevat muutamia millisekunteja myöhemmin verrattuna esimerkiksi C-kieleen. Komennot koodissa voivat myös olla epäselviä kieleen aiemmin tutustumattomille. (Lobo & Frontseca 2017)

Muita suosittuja kieliä ovat C/C++ ja Java. C on Pythonin jälkeen suosituin ohjelmointikieli 44%:n suosiolla ja 19%:n prioriteetilla, kun taas Java on samalla tasolla C:n kanssa 39%:n suosiolla ja 16%:n prioriteetilla. (Kuva 1.)

Pythoniin verrattuna C on kielenä paljon nopeampi, ja sitä usein käytetään esimerkiksi koneohjelmointiin tämän takia. Nykyaikana kieli on tosin enemmän käytössä sen takia, että sitä on käytetty monessa paikassa jo ennestään. Jotta sitä voi päivittää, C-kielen käyttö on täten pakollista. Bottien suhteen kielenä C:tä käytetään enemmän videopelihahmojen tekoälyjen rakennuksessa, kuin varsinaisissa keskusteluboteissa. (Voskaglau 2017.)

Javan käyttöä priorisoivat eniten tietokoneohjelmien tekijät, ja Voskoglou kertoo kielen kanssa olevan sama tilanne kuin C:n kanssa: Yhtiö on käyttänyt Javaa pitkään, jolloin sen muuttaminen jollekin muulle kielelle on ongelmallista. Hän lisäksi nostaa esille, että Javaa kielenä ei yleensä opetella uteliaisuudesta. (Voskaglau 2017.)

Suurin ongelma chattiboteilla tulee olemaan niiden keskustelutaidot. Botit puhuvat hyvin staattisesti, ja sanovat mitä niiden halutaan sanovan. Tätä pahentaa se, että bottien rakennuksen helpottamiseksi tuotettuja valmisohjelmia on tuotettu niin paljon, että yksinkertaisten bottien määrä on nopeasti kasvanut, ja niiden kieli on yhtä yksinkertaista. Chattibotti ei esimerkiksi ymmärrä, että monenlaisiin erilaisiin kysymyksiin voi olla sama vastaus, ennen kuin nämä kaikki vaihtoehdot on sille kerrottu. Tämä on varsinkin suomen kielessä ongelma, koska sanoilla on monta eri muotoa. Niin kuin kaikki ohjelmat, botit kuitenkin voivat parantua ajan mittaan, oppien käyttäjien määrän kasvaessa lisää, kun niille ohjelmoidaan lisää mahdollisia vastauksia asiakkaiden kysymysten mukaan. (Chatbot community 2017.)

Voskoglou kuitenkin sanoo asian artikkelissaan parhaiten: parasta kieltä koneoppimiseen ei ole olemassakaan. Kaikki perustuu siihen, millaista bottia haluat rakentaa ja mitä kieltä osaat käyttää aiheeseen perustuen. Hän kuitenkin antaa oman mielipiteensä aiheesta sanoen, että puhtaalta pöydältä aloittavalle botin kirjoittajalle paras kieli olisi Python, mutta Javan käyttöön kannattaa valmistautua, jos haluaa tehdä töitä isoja yhtiöitä varten. (Christina Voskaglau 2017.)

Foredatan sivut käyttävät tällä hetkellä Javaa ohjelmointikielenään, mutta yhteistuumin päädyimme tekemään chattibotin Pythonilla. Foredata on jo tehnyt päätöksen siirtyä pois Java-pohjaisesta ratkaisusta, jolloin artikkeli ja päätös vain vahvistavat varmuuttani Pythonin käyttöön projektia varten.

3 CHATTIBOTIN RAKENTAMINEN

3.1 Ensimmäinen suunnitelma

Opinnäytetyöni suunnitelmana on rakentaa omatekoinen botti Python-ohjelmointikielellä, joka tallentaa asiakkaan nimen ja toiveammatit, ja tulevaisuudessa se voi oppia myös suosittelemaan eri ammattialoja asiakkaan antamien tietojen mukaan. Testausvaiheessa saatamme käyttää erillistä sisäänkirjautumista tietojen suojaamiseksi, mutta varsinainen työ ei tule sitä tarvitsemaan. Asiakas pystyisi myös muuttamaan kyseisiä tietoja halutessaan. Tarkoitus olisi tehdä chattibotti sillä tavalla, että se saataisiin yhdistettyä Foredatan tietokantaan ja kahteen erilliseen sivuun, jolloin kyseisten tietojen tallennus tarvitsisi tehdä vain kerran. Tätä varten tulen tarvitsemaan tietokannan, jonne tallentaa nämä tiedot ja kaksi erillistä verkkosivua. Tällä hetkellä suunnitelmana on testata ohjelmaa omalla koneella, josta sen voi myöhemmin siirtää virallisille sivuille. Chattibotin rakennusta varten käytämme Sumit Rajin kirjassa ”Building Chatbots with Python: Using Natural Language Processing and Machine Learning” (2019) käytettyä esimerkkiä ja muita verkkolähteitä hyväksemme.

Dayn ja Tuckerin kommentit quorassa kysymykseen ”Pystyvätkö monet eri servuilla olevat verkkosivut käyttämään yhtä tietokantaa?” (2019) kertoo, että yhden tietokannan käyttö monen sivun ja botille annettujen tietojen tallennukseen on hyvinkin mahdollinen. Monet sanovat, että se on jopa perusasetus tällaiseen toimintaan. Yhden tietokannan käyttöön kuuluu silti riskejä, suurin osa perustuen tallenteiden määrään.

Riskejä Dayn ja Tuckerin mukaan ovat seuraavat:

- Tallennuksia ei pystytä lukemaan samaan aikaan, kun niitä kirjoitetaan.
- Liian monen ’tablen’ kasaaminen samaan tietokantaan voi johtaa pitkiin sivun avautumisaikoihin (korjattavissa välimuisteilla).
- Suuri määrä ihmisiä samaan aikaan hidastaa toimintaa huomattavasti.
- Kahdesta eri paikasta otetut tiedot eivät saata yhdistyä samaan paikkaan, vaikka ne käsittelevät samaa asiaa.

Suurin osa näistä ei tule olemaan ongelma suunnittelemastamme botista johtuen, jolloin esimerkiksi vältetään kahden erilaisen tallennuksen tuotto. Teoriassa botti tallentaisi tiedot vain kerran samalla algoritmilla tietokantaan, jolloin tulokset tallentuvat ilman ongelmia. Samanaikaisten käyttäjien määrä on aihe, jota emme pysty itse kontrolloimaan. Tietokannan järjestely on myös aiheena tärkeä, mutta tulee ongelmana esille vasta myöhemmin konkreettisesti.

Suurin työ tulee olemaan varsinaisen chattibotin suunnittelu ja ohjelmointi. Chattibotin suunnittelussa tiettyyn tehtävään on hyvä harkita kolmea erilaista kysymystä, jotta saadaan selville, voiko chattibotti hoitaa tehtävän ja kuinka hyvin sen tarvitsee keskittyä eri osiin.

Ensimmäinen kysymys: Voidaanko ongelma, jota varten bottia rakennetaan, hoitaa yksinkertaisella keskustelulla, tai kysymyksellä ja vastauksella? Botit pystyvät hoitamaan myös vaikeita tehtäviä, mutta aluksi on hyvä keskittyä hoitamaan asiat mahdollisimman nopeasti ja yksinkertaisesti. (Raj 2019, 17.) Vastaisin kysymykseen kyllä, koska chattibottimme tulee pyytämään tallennettavat tiedot, jolloin asiakkaan tarvitsee vain kommentoida, mitä hän haluaa tallentaa.

Toinen kysymys: Sisältääkö tehtävä paljon hyvin usein toistuvia ongelmia, jotka tarvitsevat tietojen tarkkaa analyysia tai tiedon noutamista? Tämä on kysymyksenä enemmän käyttäjiä ja palveluntarjoajia varten, koska botin tarkoituksena on poistaa näiden ihmisten tarve tehdä ’tylsiä’ töitä itse. (Raj 2019, 17.) Tämä on kysymyksenä vähän vaikeampi, sillä teoriassa bottimme käyttäisi saamiaan tietoja ammatteihin liittyen etsiäkseen muita ammatteja asiakkaalle, jolloin sen tarvitsisi tutkia saamiaan ehdotuksia joka kerta, kun uusi asiakas kirjautuu sisään, tai kun asiakas haluaa muuttaa työlistaansa. Näiden tietojen mukaan vastaisin kysymykseen kyllä.

Viimeinen kysymys: Voiko botin tekemän työn automatisoida ja korjata? Tämä kysymys on tärkeä, koska bottien toiminta ei ole hyvin persoonallista. Tämä voi hankaloittaa tilanteita, joissa tarvitaan hyvin yksilökohtaista toimintaa. (Raj 2019, 18.) Meidän bottimme tulee pääasiassa toimimaan yksinkertaisena tietojen muuttimena ja tallentimena. Näiden kaikkien kysymysten vastausten mukaan, botti sopii suunnittelemaamme työhön erittäin hyvin.

Tämän tehtyämme voimme siirtyä varsinaiseen suunnitteluosioon. Ensimmäiseksi päättelemme tarkemmin bottimme laajuutta opinnäytetyötä varten. (Raj 2019, 65.) Tämän ei tietenkään tarvitse olla kaikki, mitä botin pitäisi osata, mutta nämä ovat prioriteetit, jotka haluaisimme botin ainakin osaavan.

Valmis bottimme osaa

- tervehtiä, kun käyttäjä tervehtii sitä
- ymmärtää, kun käyttäjä haluaa kirjautua sisään
- havaita, jos salasana on väärä
- havaita, jos käyttäjää ei ole olemassa
- pyytää käyttäjätunnusta ja sen salasanaa
- pyytää käyttäjän nimen ja työtaidot tietokantaan tallennusta varten
- muuttaa käyttäjän tietoja pyynnöstä
- tarjota työpaikkaa perustuen käyttäjän taitoihin

Tästä listasta voimme päätellä bottimme harjoitusta varten asiakkaiden aikomuksia. Näitä harkitaan valmiiksi, jotta tiedämme tarkasti, mitä meidän pitää botillemme opettaa. (Raj 2019, 66.) Laajuudesta voimme päätellä esimerkiksi seuraavat aikomukset:

- käyttäjä tervehtii chattibottia
- käyttäjä haluaa kirjautua sisään
- käyttäjä haluaa tehdä itselleen käyttäjän sivustolle
- käyttäjä haluaa katsoa tietojaan
- käyttäjä haluaa muuttaa tietojaan
- käyttäjä haluaa työpaikkaehdotuksen.

Lopuksi teorisoimme, millainen olisi niin sanotusti normaali keskustelu botin ja käyttäjän välillä. Tämä tarjoaa hyvän esimerkin siitä, miten haluamme botin visuaalisesti toimivan ja näyttävän, miten se voi siirtyä eri tilanteista seuraavaan. (Raj 2019, 66.)

Käyttäjä: Hei.
Chattibotti: Päivää! Olen Foreammatin chattibotti. Haluatko kirjautua sisään?
Käyttäjä: Kyllä.
Chattibotti: Anna käyttäjätunnuksesi.
Käyttäjä: Matti Meikäläinen
Chattibotti: Anna salasanasi.
Käyttäjä: *****
Chattibotti: Antamasi salasana on väärin… Haluatko yrittää uudelleen?
Käyttäjä: Kyllä.
Chattibotti: Anna salasanasi.
Käyttäjä: *****
Chattibotti: Tervetuloa Matti Meikäläinen!
Chattibotti: Sinun tämänhetkiset työtaitosi ovat: kokkaaminen, ajaminen, rakentaminen ja siivoaminen.
Käyttäjä: Haluan muuttaa tietojani.
Chattibotti: Haluatko muuttaa nimeäsi?
Käyttäjä: En.
Chattibotti: Haluatko muuttaa työtaitojasi?
Käyttäjä: Kyllä.
Chattibotti: Kirjoita työtaitosi.
Käyttäjä: kokkaaminen, ajaminen, rakentaminen, siivoaminen ja opettaminen.
Chattibotti: Työtaitosi on muutettu. Haluatko kirjautua ulos?
Käyttäjä: Kyllä.
Chattibotti: Olet kirjautunut ulos. Näkemiin!

Näillä esimerkeillä, voimme alkaa rakentamaan varsinaista bottia. Koska meillä on selvä kuva siitä, miten botin pitää toimia, meidän tarvitsee nyt vain seurata esimerkkejä ja käyttää niiden antamia tietoja hyväksemme. Osasta näistä on hyötyä vasta, kun olemme saaneet varsinaisen pohjan rakennettua.

3.2 Foredata-yrityksen mielipide

Näytin 28.2.2020 Jari Järviselle tämänhetkisen suunnitelmani, ja kysyin hänen mielipidettään tarvittavista muutoksista tai lisäyksistä. Hänen tarpeensa botin suhteen oli, että botti tallentaisi käyttäjän työhistorian ja pystyisi nostamaan sen esille pyydettäessä. Ongelmalliseksi hän nosti esille työpaikan ehdottamisen. Hänen mukaansa, jotta sellainen toiminta saataisiin rakennettua, botti tarvitsisi integroida systeemiin kunnolla, jolloin se pystyisi käyttämään sivuston muita tietoja hyväkseen. Tämän takia hän suositteli työpaikan ehdottamista vain asiakkaan itse antamien tietojen mukaan.

Muita ongelmia, joita hän toi esille, olivat botin testaus yksityisesti ja integrointi muissa käyttöjärjestelmissä toimivaksi. Jos chattibotti testausvaiheessa esimerkiksi kytketään Facebook-sivustoon, kuka tahansa voi teoriassa mennä kokeilemaan sitä, jos he sen löytävät. Integrointia varten hänellä oli tarjota suunnitelmaksi ”dockerointi”, eli asentaisimme chattibotin virtuaalikoneisiin, joissa olisi eri käyttöjärjestelmät. Prosessina se kuitenkin voi olla kallis, minkä takia hän suositteli sen tekemistä tutkimukseni jälkeen.

Hänellä oli myös toiveita koskien botin toimintaa: sen sijaan, että asiakas itse täyttäisi tarvittavat tiedot, botti tarjoaisi asiakkaille vaihtoehtoja, joista valita heidän työkokemuksensa ja koulutuksensa. Tämä auttaisi Foredataa osaamiskartoituksen tekemisessä, ilman että tulos olisi vaikealukuinen. Botin rakenteen suhteen hän haluaisi, että sen pystyisi asentamaan kuin puhelin-appin heidän sivustolleen heti käyttövalmiina.

3.3 Esimerkki-botin rakennus

Ennen kuin aloin rakentamaan bottiani virallisesti, tein Sumit Rajin kirjassa olevan esimerkkibotin. (Raj 2019, 105-154.) Tällä tavalla sain valmiin koodin, jota voin soveltaa omaani. Lisäksi tarkkojen ohjeiden seuraaminen helpotti varsinaista oppimisprosessiani. Tulen myös käymään läpi eri koodin osia, ja niiden tarkoitusta. Rakennusprosessissa tulen käyttämään JSON:ia, Anacondaa, ja sen sisältämää Pythonin kirjoitusohjelmaa nimeltä ‘Spyder’. Minulla on tämän ohjelman käytöstä eniten ko-

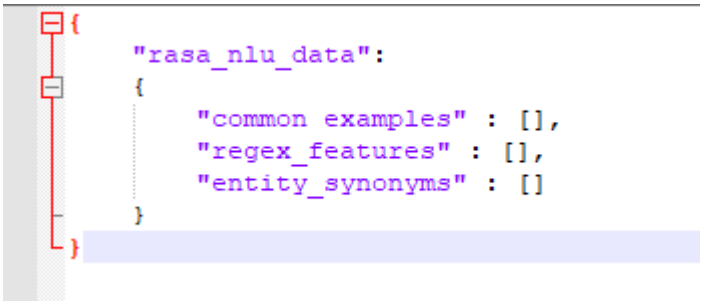
kemusta verrattuna muihin ohjelmiin ja monta esimerkkiä muusta python toiminnasta, joita voin käyttää hyväkseni tätä tehdessäni.

Esimerkissä ja omassa tuotoksessani tullaan käyttämään Avoimen lähdekoodin kirjastoa (Open source library) nimeltä Rasa NLU, mitä kirjassa suositellaan opettelua varten. Se ei käytä hyväkseen pilvipalvelua niin kuin monet valmiit botin rakennuspohjat, mikä varmistaa myös sen, että bottini käyttää tallennusprosessiin vain yhtiön omaa palvelinta. Rasa on verkkokirjastona myös hyvin aktiivinen, jolloin pystyn hankkimaan verkosta ajankohtaista apua tarvittaessa. Botin rakentaminen alusta alkaen antaa minulle myös vapaat kädet toteuttaa se täysin sellaiseksi kuin haluan. (Raj 2019, 107.)

Harjoitusbottiani varten valitsin Pipelinen. Pipeline on joukko erilaisia algoritmeja, joilla voin opettaa bottiani. Tätä varten voin valita kahdesta erilaisesta Pipelinesta: Spacy_sklearn tai Tensorflow_embedding. Molemmilla on omat hyvät ja huonot puolensa, mutta tätä varten käytän Spacy_sklearnia.

Esimerkkibottini tulee antamaan asiakkaalle heidän horoskooppinsa, jolloin tarvittavan datan määrä on pieni. Täten Spacy_sklearn sopii tämän botin tarkoitukseen parhaiten. Botin taustalle on tehty keskustelusta aikeet ja esimerkki jo valmiiksi, mutta keskityn tässä tekstissä pääosin ohjelmoinnin ja tekniikan puoleen. Mainitsen niistä tarvittaessa kontekstin saamiseksi.

Tässä vaiheessa rakennan Horoscope_bot kansion, jonne teen toisen kansion Data, ja sinne rakennan JSON-kielisen tiedoston. Rasa_nlu pystyy siitä lukemaan tarvitsemansa aikeet ja oliot, mutta pystyisi myös lukemaan ne isommasta hakemistosta verkon kautta. Data.json tiedosto ennen muuta lisättyä koodia näyttää tältä:



Kuva 2: JSON-koodin pohja ennen lisäyksiä.

Common examples avain on bottini varsinainen harjoituspaikka ja kaikki esimerkkimme menevät sen alle. Regex_features taas auttaa bottiani tunnistamaan aikomuksia ja olioita mitä Common Examplesissa käytetään. Tämän jälkeen voin lisätä common examplesiin seuraavat:

```
"rasa_nlu_data":
{
    "common examples" : [
        {
            "text": "Hello",
            "intent": "greeting",
            "entities": []
        },
        {
            "text": "I want to know my horoscope",
            "intent": "get_horoscope",
            "entities": []
        },
        {
            "text": "can you please tell me my horoscope",
            "intent": "get_horoscope",
            "entities": []
        },
        {
            "text": "Please subscribe me",
            "intent": "subscription",
            "entities": []
        }
                                ],
    "regex_features" : [],
    "entity_synonyms" : []
}
```

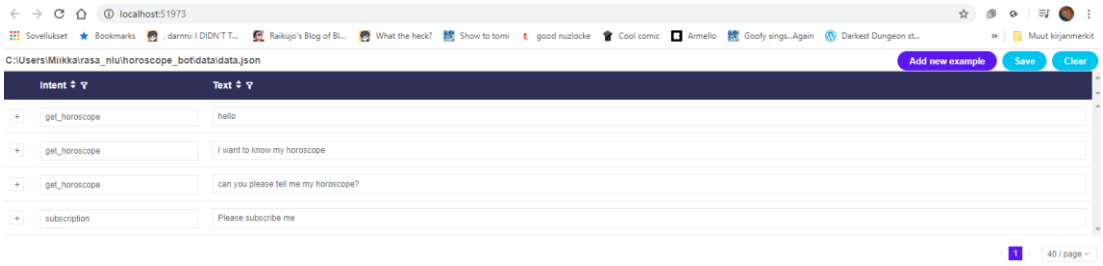
Kuva 3: Json koodin jatkoa

Niin kuin kuvasta voi huomata, olen kirjoittanut ylös erilaisia lauseita ja mihin aikeeseen se liittyy. Lisäksi minulla on tilanteeseen liityviä olioita varten paikka valmiina.

Kun sain tämän koodin rakennetuksi, siirryin asentamaan koneelleni Node.Js:n seuraavaa askelta varten. Tarkoituksena oli asentaa rasa-nlu-trainer, joka auttaisi visualisoimaan paremmin varsinaisen botin opetusprosessin. Asentaessani komennon laitteeseen se kuitenkin varoitti, että ohjelma on vanhentunut, ja sitä ei enää päivitetä.

Se kuitenkin asentui, mikä on hyvä asia tehtävän tarkkaan seuraamista varten, mutta panin merkille githubin ehdottaman korvaavan vaihtoehdon rasa x kaiken varalta.

Käyttäen rasa-nlu-traineria, päätin rakentaa common examples osion kokonaan uudestaan. Lopputuloksena komento rasa-nlu-trainer tuottaa seuraavan sivuston paikalliselle palvelimelle ja avaa sen selaimeen katsottavaksi ja muutettavaksi.



Kuva 4: Rasa-nlu-trainer komennon avaama sivu.

Tämä ohjelma helpottaa hyvin paljon opetusprosessiani: voimme ohittaa koodauksen hankaluuden ja vain toteuttaa sovelluksen tarvitseman toiminnallisuuden. Ohjelma antaa helposti poistaa koodista sellaisia tietoja, mitä ei enää tarvita.

Seuraavaksi lisätään horoscope-bot kansioon aiemmin mainittu Pipeline. Tätä varten rakennan JSON filen nimeltä config.json, ja lisään sinne seuraavan osan koodia:

```
{
    "pipeline":"tensorflow_embedding",
    "path":"./models/nlu",
    "data":"./data/data.json"
}
```

Kuva 5: config.json

Pipeline kertoo, mitä pipelinea käytän kaikkien tietojen keräämiseen. Path kertoo, missä pidän bottimme mallia. Tässä koodissa kohteessa ”./models/nlu”. data taas kertoo missä harjoitusdata on. Tässä koodissa käytetään aiempaa JSON tiedostoa, minkä kirjoitin.

Tässä kohdassa ohjelmoin ensimmäisen python-pohjaisen koodin bottia varten. (kuva 6)

rasa-nlu.ipynb

```
1 from rasa_nlu.training_data import load_data
2 from rasa_nlu.model import trainer
3 from rasa_nlu import config
4 from rasa_nlu.model import interpreter
5
6 def train_horoscopebot(data_json, config_file, model_dir):
7     training_data = load_data(data_json)
8     trainer = Trainer(config.load(config_file))
9     trainer.train(training_data)
10    model_directory = trainer.persist (model_dir, fixed_model_name = 'horoscopebot')
11
12 def predict_intent(text):
13    interpreter = interpreter.load('./models/nlu/default/horoscopebot')
14    print(interpreter.parse(text))
```

Kuva 6: Rasa-nlu.ipynb

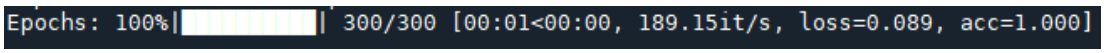
Ensimmäiset neljä koodinpätkää käsittelevät tarvittavien kirjastojen hakua rasa_nlu paketista. Sitten rakennettiin metodit train_horoscopebot, joka nimensä mukaisesti harjoittaa bottia datan ja config-tiedoston perusteella ja tallentaa ne model directoryyn, ja predict_intent, joka käyttää interpreter mallia ladatakseen verkosta valmiita mallitiedostoja ja antaa käyttäjälle mahdollisuuden ennustaa, mitä muita on käytössä.

Kokeiltuani saada tätä koodia itsessään toimimaan, päätin etsiä apua verkosta. Löysin medium.comista tutoriaalin, joka seurasi suurin piirtein samoja ohjeita, mitä kirja tarjosi. Rakenteellisesti tämän sivun tarjoama botti oli hyvin samanlainen kirjamme botin kanssa. Erona pääasiassa oli se, että se käytti Rasan versiota 12.3 13.3:n sijaan. Lisäksi Medium versio ei käyttänyt .ipynb tiedostoja, minkä pitäisi helpottaa toimintaani. Bottini myös käytti spacyä tensorflowin sijaan, johtaen siihen, että config tiedostoni on hieman erilainen ja bottini python koodin tarkoitus on uusi.

Harmikseni tätä koodia yrittäessä sain selville, että koodi oli vanhentunut, ja täten käyttökelvoton. Tämä tarkoittaisi, että minun pitäisi soveltaa koodia muunlaiseksi tai aloittaa kokonaan alusta. Mutta tätä koodia käyttämällä sain ainakin korjattua ison ongelman, mikä minulla oli aiemman koodin kanssa: sain python kirjastoni toimimaan.

Tämän jälkeen päätin siirtyä takaisin kirjan antamaan bottiin ja koodiin. Kirja on uudempi, mistä syystä otaksun, ettei siinä ole samaa ongelmaa kuin aiemmassa botissa, ja lisäksi olen löytänyt suurimman osan ongelmakohdista, joita minulla aiemmin oli. Koska olen nyt tehnyt tämän pari kertaa, olen onnistunut saamaan rakennettua koodit hyvin nopeasti. Koodi näyttää tällä hetkeltä samalta kuin kappaleen alussa.

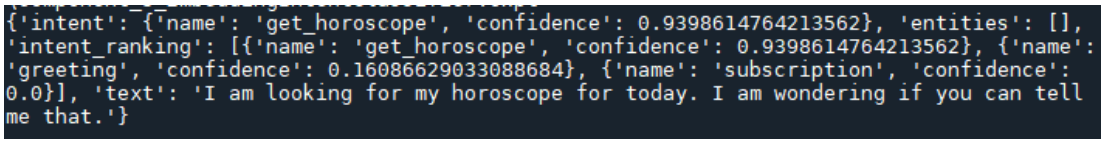
Ongelmien suhteen otin yhteyttä opinnäytetyön ohjaajaani. Tunnin pähkäilyn ja etsimisen jälkeen löysimme ongelmaan syyn: Rasa nlu.ipynb filemme nimi aiheutti sen, että kun koodissani haettiin rasa_nlu kirjastoa koodia varten, se sen sijaan yritti ottaa sen itsestään kirjaston sijaan. Tämän takia vaihdoin rasa_nlu.ipynb:n nimen Ralu.ipynb:ksi sen sijaan. Nyt koodi antaa meidän kutsua kyseisestä koodista komentoja.



Kuva 7: Train_horoscopebotin tulos

Tämä tuottaa minulle myös models-kansion. Sinne menevät kaikki meidän index-, meta- ja pickle- tiedostomme.

Seuraavaksi testasin predict_intent komentoa. Tämä testaa, millä tasolla bottimme tunnistaa eri aikeita sille annetun datan mukaan. Komennon antaessamme, se antaa seuraavaa:



Kuva 8: Predict_intentin tulos

Tämä kertoo, että se tunnisti siltä odotetun aikomuksen ‘get_horoscope’ kuvan antamalla varmuudella eli tässä kuvassa ‘confidence’. Tämä on tietysti osittain sen takia, että meidän intent määrämme on suhteellisen pieni. Tämän jälkeen kuitenkin minulla on opetusosuus botistamme valmiina. Tämä toimii monen varsinaisen bottiohjelman taustalla, eikä sitä yleensä nähdä käytössä. Tätä pitää käyttää aina, kun bottiin lisätään uusia intenttejä, jotta tiedetään, että botti löytää ja tunnistaa ne.

Seuraavaksi esitetään aiemmin käsittelemiäni aiheita, kuten uusien toimintojen rakentamisesta bottia varten, keskustelupohjia, slotteja ja käytettäviä olioita. Slotteihin laitetaan, missä muodossa tieto on (esimerkiksi bool tai text). Aikomuksia olemme käyttäneet jo aiemmin. Ne tarkoittavat, mitä kysyjä saattaa haluta. Oliot ovat erilaisia parametrejä, mitä käytetään ohjelman toiminnan aikana. Pohjien avulla voidaan päättää, mitä bottimme varsinaisesti sanoo tunnistaessaan aikeen ja toiminnot ovat, mitä botti tekee eri tilanteissa. Rakennan bottiamme varten Yaml-pohjaisen domain-tiedoston, mikä käyttää näitä kaikkia hyväkseen.

Jokaiselle pohjalle (template) on tehty myös action (toiminto). Ensimmäiset neljä toimintoa ovatkin pääasiassa niihin liittyvien pohjien lausahduksien sanominen. Get_todays_horoscope ja subscribe_user komennot kuitenkin vaativat sen, että rakennan niitä varten uuden toiminnon koodia käyttämällä. Voin sijoittaa molemmat samaan koodiin nimeltä actions.py:

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import requests

from rasa_core_sdk import Action
from rasa_core_sdk.events import SlotSet

class GetTodaysHoroscope(Action):

    def name(self):
        return "get_todays_horoscope"
    def run(self, dispatcher, tracker, domain):
        #type: Dispatcher, DialogueStateTracker, Domain) -> list[Event]

        user_horoscope_sign = tracker.get_slot('horoscope_sign')
        base_url = http://horoscope-api.herokuapp.com/horoscope/{day}/{sign}
        url = base_url.format(**{'day': "today", 'sign': user_horoscope_sign})

        res = requests.get(url)
        todays_horoscope = res.json()['horoscope']
        response = "Your today's horoscope:\n{}".format(todays_horoscope)

        dispatche.utter_message(response)
        return [SlotSet("horoscope_sign", user_horoscope_sign)]
```

Kuva 9: Actions.py

Tämä koodi hakee verkkosivulta http://horoscope-api.herokuapp.com/horoscope/{day}/{sign} päivän horoskoopin, ja sitten tulostaa sen muodossa “Your today’s horoscope: diibadaaba”. pystyn samalla tavalla otta-

maan verkosta tarvittavat tiedot vaihtamalla kohdesivua. Samaan koodiin lisään myös subscribeuser toiminnon:

```
class SubscribeUser(Action):
    def name(self):
        return "subscribe_user"

    def run(self, dispatcher, tracker, domain):
        #type: (Dispatcher, DialogueStateTracker, Domain) -> List[Event]

        subscribe = tracker.get_slot('subscribe')

        if subscribe == "true":
            response = "You're successfully subscribed"
        if subscribe == "false":
            response = "You're successfully unsubscribed"

        dispatcher.utter_message(response)
        return [SlotSet("subscribe", subscribe)]
```

Kuva 10: SubscribeUser action

Tämä on vähän yksinkertaisempi toteutus. Koodi tunnistaa onko asiakas sanonut kyllä vai ei subscribe-pyyntöön, ja sen jälkeen palauttaa vastauksen perusteella jommankumman kahdesta vaihtoehdosta.

Tämän jälkeen kirjoitan stories.md tekstitiedoston. Tämä antaa botillemme esimerkin siitä, miten keskustelu voi mennä asiakkaan kanssa käyttäen ohjelmoimiamme toimintoja.

```
## story_001
* greeting
    - utter_greet
* get_horoscope
    - utter_ask_horoscope_sign
* get_horoscope{"horoscope_sign": "Capricorn"}
    - slot{"horoscope_sign": "Aries"}
    - get_todays_horoscope
    - utter_subscribe

## story_002
* greeting
    - utter_greet
* get_horoscope{"horoscope_sign": "Capricorn}
    - slot{"horoscope_sign": "Cancer}
    - get_todays_horoscope
    - utter_subscribe
* subscription
    - slot{"subscribe": "True"}
    - subscribe_user

## Horoscope query with horoscope_sign
* greeting
    - utter_greet
* get_horoscope
    -utter_ask_horoscope_sign
* get_horoscope{"horoscope_sign": "capricorn"}
    - slot{"horoscope_sign": "capricorn"}
    - get_todays_horoscope
    - slot{"horoscope_sign": "capricorn"}
    - utter_subscribe
* subscription{"Subscribe": "True"}
    - slot{"subscribe": "True"}
    - subscribe_user
    - slot{"subscribe": true}
```

kuva 11: Stories.md osa 1

```
## Horoscope with sign provided
* greeting
    - utter_greet
* get_horoscope
    -utter_ask_horoscope_sign
* get_horoscope{"horoscope_sign": "leo"}
    - slot{"horoscope_sign": "leo"}
    - get_todays_horoscope
    - slot{"horoscope_sign": "leo"}
    - utter_subscribe
* subscription{"Subscribe": "True"}
    - slot{"subscribe": "True"}
    - subscribe_user
    - slot{"subscribe": true}

## When user directly asks for subscription
* greeting
    - utter_greet
* subscription{"subscribe": "True"}
    - slot{"subscribe": "True"}
    - subscribe_user
    -slot{"subscribe": true}
```

Kuva 12: Stories.md osa 2

Tämä on helppoa tulkita, koska jo toimintojen nimet kertovat, mitä koodin mukaan pitäisi tapahtua. Erona voidaan kuitenkin huomata ensimmäisen ja toisen tarinan välillä se, että toisessa tarinassa botin käyttäjä tilaa päivittäisen horoskoopin. Tähän tiedostoon voidaan aina kirjoittaa lisää tarinoita tarvittaessa.

Työssä kerrotaan myös toisesta tavasta, millä rakentaa tällaisia tarinoita ilman, että niitä tarvitsisi kirjoittaa erikseen yksi kerrallaan. Rasalla on ‘interactive learning’ työkalu, mikä antaa meidän tehdä niitä lisää ilman sitä. Tätä varten kirjoitan train_initialize.py tiedoston:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

from rasa_core import utils
from rasa_core.agent import Agent
from rasa_core.policies.keras_policy import KerasPolicy
from rasa_core.policies.memoization import MemoizationPolicy
from rasa_core.policies.sklearn_policy import sklearnpolicy

if __name__ == '__main__':
    utils.configure_colored_logging(loglevel="DEBUG")

    training_data_file = './data/stories.md'
    model_path = './models/dialogue'
    agent = Agent("horoscope_domain.yml",
                  policies=[MemoizationPolicy(), KerasPolicy()])

    training_data = agent.load_data(training_data_file)

    agent.train(
        training_data,
        augmentation_factor=50,
        epochs=500,
        batch_size=10,

        validation_split=0.2
        )

agent.persist(model_path)
```

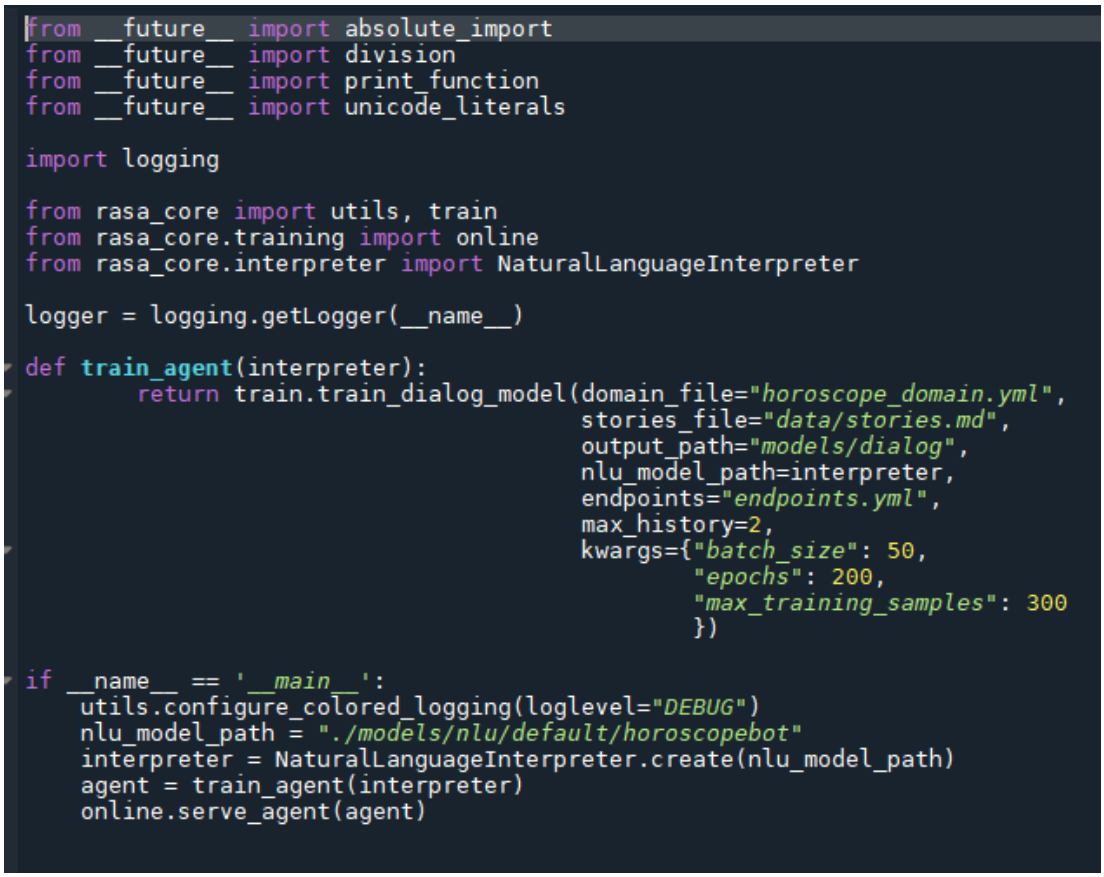
kuva 13: Train_initialize.py

Seuraavaksi päästään keskustelemaan varsinaisen botin kanssa, ja varmistamaan, että se reagoi oikein eri kysymyksiin. Tätä varten rakennan endpoints.yml tiedoston ja train_online.py tiedoston. Endpoints_yml:n tehtävänä on nostaa rasa methodimme esille http verkkoa varten. (We can expose the Rasa method as http API:s)

```
action_endpoint:
    url: http://localhost:5055/webhook

#nlg:
#   url: http://localhost:5056/nlg

core_endpoint:
    url: http://localhost:5005
```

kuva 14: Endpoints.yml

Train_online.py taas on nimensä mukaisesti botin harjoitusta varten.



```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import logging

from rasa_core import utils, train
from rasa_core.training import online
from rasa_core.interpreter import NaturalLanguageInterpreter

logger = logging.getLogger(__name__)

def train_agent(interpreter):
        return train.train_dialog_model(domain_file="horoscope_domain.yml",
                                        stories_file="data/stories.md",
                                        output_path="models/dialog",
                                        nlu_model_path=interpreter,
                                        endpoints="endpoints.yml",
                                        max_history=2,
                                        kwargs={"batch_size": 50,
                                                "epochs": 200,
                                                "max_training_samples": 300
                                                })

if __name__ == '__main__':
    utils.configure_colored_logging(loglevel="DEBUG")
    nlu_model_path = "./models/nlu/default/horoscopebot"
    interpreter = NaturalLanguageInterpreter.create(nlu_model_path)
    agent = train_agent(interpreter)
    online.serve_agent(agent)
```

Kuva 15: Train_online.py

Nyt käytän myös aiemmin asentamaani rasa-core-sdk kirjastoa. Tämä kirjasto sisältää komentoja, joita tulen tarvitsemaan botin toimintaa varten, esimerkiksi subscribe_user ja get_todays_horoscope. Kun nämä toiminnot aktivoidaan, se antaa metodin mukaisen vastauksen. Tämän aktivointia varten pistän seuraavan komennon cmd:hen siinä kansiossa, missä action.py:mme on.

```
python -m rasa_core_sdk.endpoint --actions actions
```

Kuva 16: Komento

Ongelmaksi komennon kanssa kuitenkin nousi koodissa oleva ongelma. Base_url olio koodissamme palauttaa invalid syntax ilmoitusta. Tätä varten päätin katsoa, olinko vain kirjoittanut sen väärin kirjasta. Kokeilin asettaa sivuston hakasulkujen sisään, jolloin saatiin kirjassa kuvattu tilanne.

```
(base) C:\Users\Miikka\Documents\rasa_nlu\horoscope_bot>python -m rasa_core_sdk.endpoint --actions actions
2020-05-19 13:44:43          __main__  - Starting action endpoint server...
2020-05-19 13:44:43          rasa_core_sdk.executor  - Registered function for 'get_todays_horoscope'.
2020-05-19 13:44:43          rasa_core_sdk.executor  - Registered function for 'subscribe_user'.
2020-05-19 13:44:44          __main__  - Action endpoint is up and running. on ('0.0.0.0', 5055)
```

Kuva 17: Komennon tulos

Tämä rakentaa serverin localhostiin, joka käyttää näitä komentoja. Tästä voi olla hyötyä muita kieliä käyttäessä, kun ei haluta sekoittaa käytettyjä funktioita toisiinsa.

Tämän jälkeen yritin käynnistää train_online.py:n. Ongelmaksi nousi jälleen vanhentunut koodi. Kun yritin importata onlinen rasa core.trainingista, se antoi ilmoituksen, että se ei löydy. Tästä syystä muutin rasa_core.trainingin rasa_core.training.interactive:ksi. Tämä ei kuitenkaan toiminut ja johti samaan ongelmaan.

Löysin kuitenkin myös toisenlaisen koodin, mikä kertoi, että online oli kokonaan muutettu interactiveksi. Muutin rasa_core.training.interactiven rasa_core.training import interactiveksi, ja tämän johdosta muutin myös train_online.py:n serve agent lineen onlinen sijaan interactiven. Tämä sai asiaa eteenpäin, vaikka johtikin seuraavaan ongelmaan. (kuva 18)

```
  File "C:\Users\Miikka\Documents\rasa_nlu\horoscope_bot\train_online.py", line 15, in
train_agent
    return train.train_dialog_model(domain_file="horoscope_domain.yml",

AttributeError: 'function' object has no attribute 'train_dialog_model'
```

Kuva 18: Train_dialog_model ongelma

Ilmoitus tarkoittaa, että funktiossa ei ole kyseistä attribuuttia. Koetin korjata aluksi ongelmaa asentamalla sdk:n aiemman version, mikä oli kerrottu kirjassa. Koska sitä ei löytynyt, päätin lisätä funktion suoraan domain tiedostoon. Se ei myöskään tuottanut muutosta. Suurin osa esimerkeistä sisälsi samat koodiin kirjoittamani asiat.

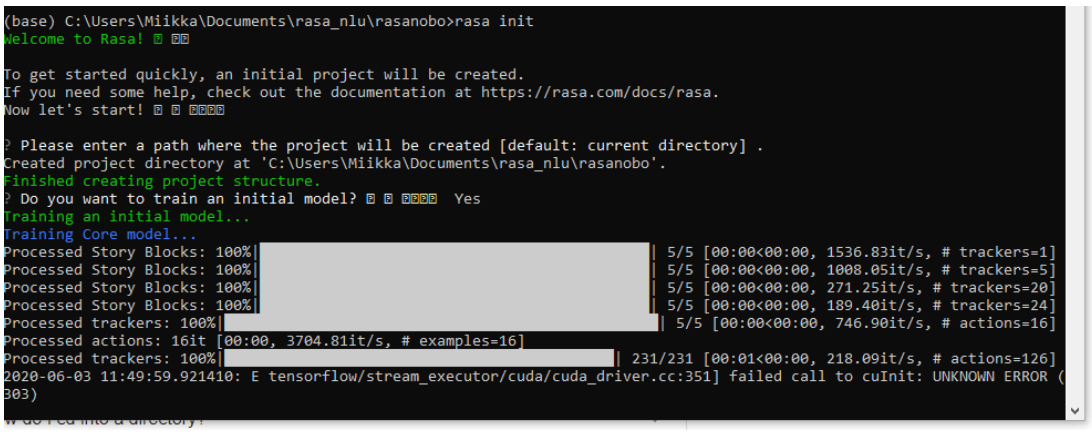
Tulin siihen tulokseen, että materiaalini, sekä kirja että verkkosivusto, olivat liian vanhentuneita opinnäytetyötäni varten. Suurin osa ongelmista on perustunut enemmän materiaalissa olevan koodin korvautumiseen kuin varsinaisiin ohjelmointiongelmiin.

4 BOTTI-SOVELLUKSEN RAKENTAMINEN

Tällä kertaa käytin esimerkkinä rasa.comin koodauksia, mikä varmisti, että kaikki käytettävä koodi olisi ajantasaista. Ainoaksi ongelmaksi voi nousta koodin päivittyminen sen kirjoittamisen aikana. Koodin tekijöiden sivusto kuitenkin sisältää kaikki muutokset, mitä koodiin on tehty. Tämä versio botista oli alusta lähtien toimiva, joten jatkoin sillä työn loppuun.

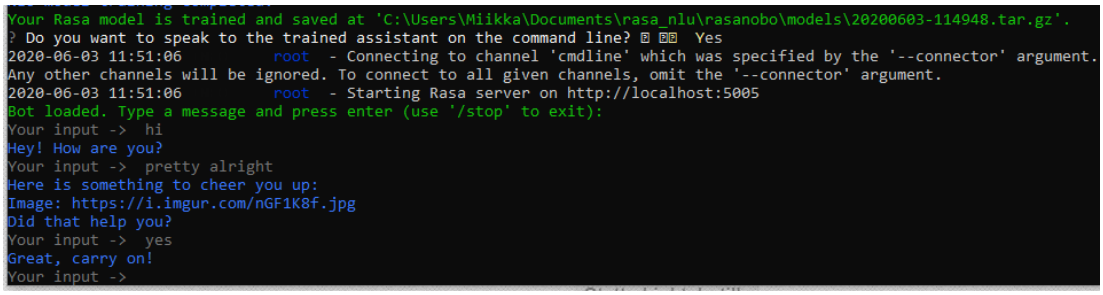
4.1 Rasa.comin esimerkki

Tämän botin yksinkertaisinta versiota varten me emme tarvitse rasa_corea, rasa_nlua tai rasa-core-sdk:ta. Ainoa mikä ladataan pohjalle, on varsinaisen Rasan versio 1.10.1. Sivusto jopa antaa meille valmiin esimerkkibotin, jota voin verrata aiemmin kehitettyyn bottiin. Teen tätä varten rasa_nlu kansioon uuden kansion nimeltä ’rasanobo’, ja siirryttyäni sinne kirjoitan komennon ’rasa init’.



Kuva 19: Rasa.init

Kuvasta nähdään, että esimerkkibotti voidaan samalla harjoittaa. Asennuksen ja harjoituksen jälkeen se antaa valmiin botin, jonka kanssa kommunikoida.



Kuva 20: Rasa botin ensimmäiset sanat.

Toteutus osoittautui aiempiin versioihin nähden helpoksi, vaikkakin kyseessä on esimerkkibotti. Tämän botin datakansio sisältää nlu.md ja stories.md tiedostot. Stories.md näyttää toimivan samalla tavalla kuin aiemmissa koodauksissamme, mutta nlu.md on näistä mielenkiintoisempi:

stories.md | endpoints.yml | config.json | Policies.yml | domain.yml | nlu.md | stories.md

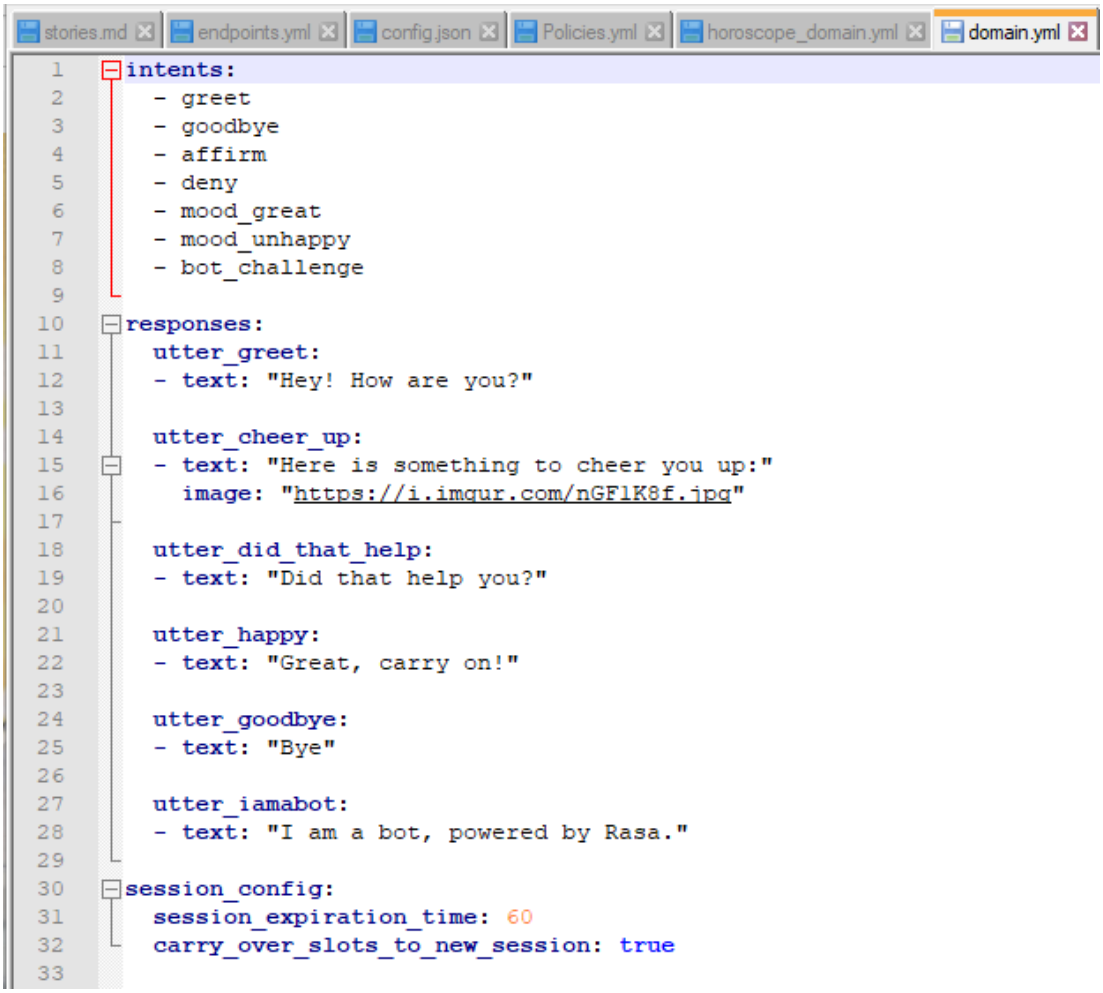
```
## intent:greet
- hey
- hello
- hi
- good morning
- good evening
- hey there

## intent:goodbye
- bye
- goodbye
- see you around
- see you later

## intent:affirm
- yes
- indeed
- of course
- that sounds good
- correct

## intent:deny
- no
- never
- I don't think so
- don't like that
- no way
- not really

## intent:mood_great
- perfect
- very good
- great
- amazing
- wonderful
- I am feeling very good
- I am great
- I'm good
```

Kuva 21: Nlu.md

Nlu.md selvästi sisältää erilaisia vastauksia, mitä botille voidaan antaa. Voin myös helposti huomata, että antamani vastaus botille ei ole listalla, mutta se silti antoi minulle tulosteen, joka vastasi aikomusta ‘deny’. Selvästi se on koodattu siten, että vastauksella on pohjana jokin vastaus, jos se ei tunnista annettua tekstiä.

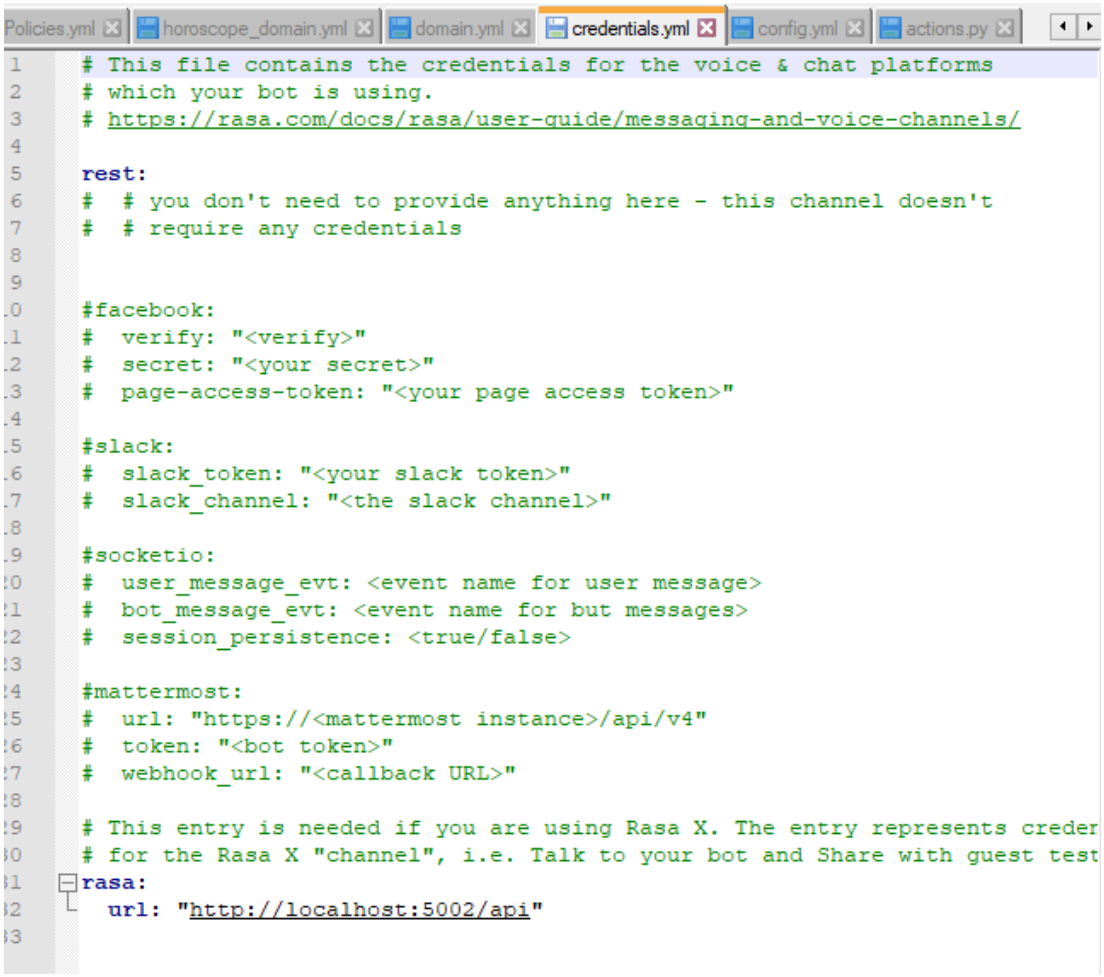
Muita yhtäläisyyksiä aiemman koodini kanssa on endpoints.yml, joka on paljon suurempi tiedosto kuin mitä toisessa botissani oli, mutta teksti siinä on kommentoitu. Domain tiedosto sisältää intentit ja mielenkiintoisesti kohdan ’responses’ templaten sijaan. Entities, slots ja actions kohdat myös puuttuvat, mutta se sisältää kohdan ’ses-

sion config'. Otaksun että tämä tarkoittaa, että botti sulkee itsensä minuutin jälkeen, jos sille ei anna syötettä.



Kuva 22: Domain.yml

Config.yml on suurin piirtein sama kuin aiemmissa koodeissa, ja actions.py toimii myös samalla tavalla. Viimeisenä tiedostona on myös credentials, mikä sisältää tieto- ja esimerkiksi facebookiin ja äänitietojen laittoa varten.



Kuva 23: Credentials.yml

Hetken tätä ajateltuani päätin, että ryhdyn tekemään bottia tämän sivuston antamien tietojen mukaan. Koska tämä on kielen virallinen sivusto, sillä ei ole vaaraa, että se olisi jäljessä. Lisäksi se antaa minulle mahdollisuuden oppia askel askeleelta mitä teen, samalla kun rakennan varsinaista bottiamme.

4.2 Oman botin rakennusta

Rasa.comissa oli selvät erilliset ohjeet, miten botti rakentuisi niistä osista mitkä juuri esittelin, joten kirjoitin omat koodini. Tämän mukaisesti botin kirjoitus alkoi suhteellisen yksinkertaisesti. Sivu itse suositteli, että jotkut osat vain kopioitaisiin esimerkistä, ja configin kanssa päätin tehdä niin.

```yaml
language: en

pipeline:
  - name: WhitespaceTokenizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 100

policies:
- name: MemoizationPolicy
  max_history: 1
- name: MappingPolicy
```

Kuva 24: Config.yml

```yaml
intents:
    - greet
    - ask
    - save
    - show
    - seeya


responses:
    utter_greet:
        - 'hello hello!'
    utter_ask:
        - 'Do you have any previous info saved?'
    utter_save:
        - 'Well, time to gather info then!'
    utter_show:
        - 'this is what we have currently'
    utter_seeya:
        - 'See ya!'
```

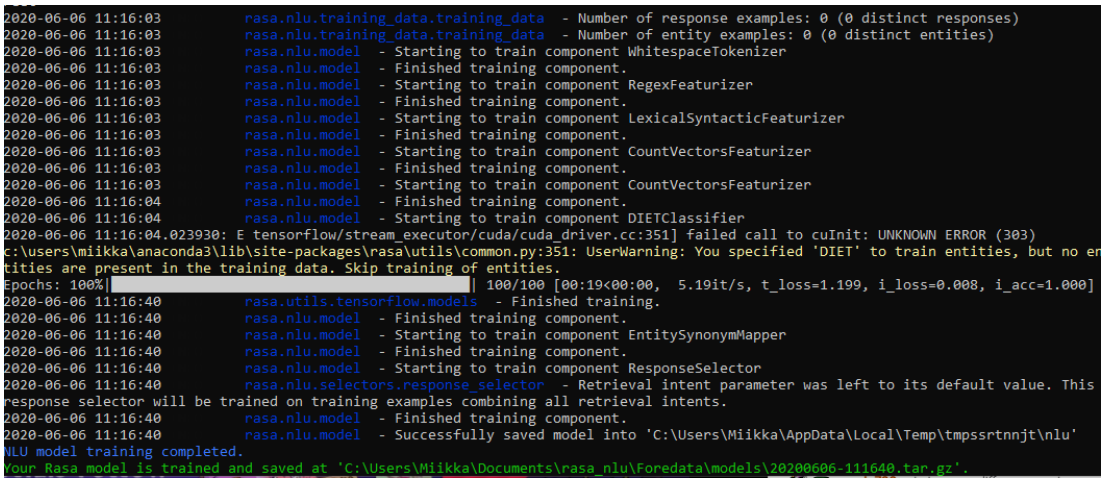
Kuva 25: Domain.yml

```
## intent:greet
- Hi
- Hey
- Hi bot
- Hey bot
- Hello
- Good morning
- hi again
- hi folks

## intent:ask
- I'd like to do something.
- What do?
- What do you have?
- Whatcha got?
- What can you do?

## intent:save
- No
- Nah
- Nope
- Guess not

## intent:show
- Yes
- Yeah
- Yup
- Indeed

## intent:seeya
- Thanks
- Thank you
- Nice job
- Thanks bot
- bye bye
- see ya later
- goodbye
- bye bot
```

```
## greet
* greet
  - utter_greet

## ask
* ask
  - utter_ask

## save
* save
  - utter_save

## show
* show
  - utter_show

## seeya
* seeya
  - utter_seeya
```

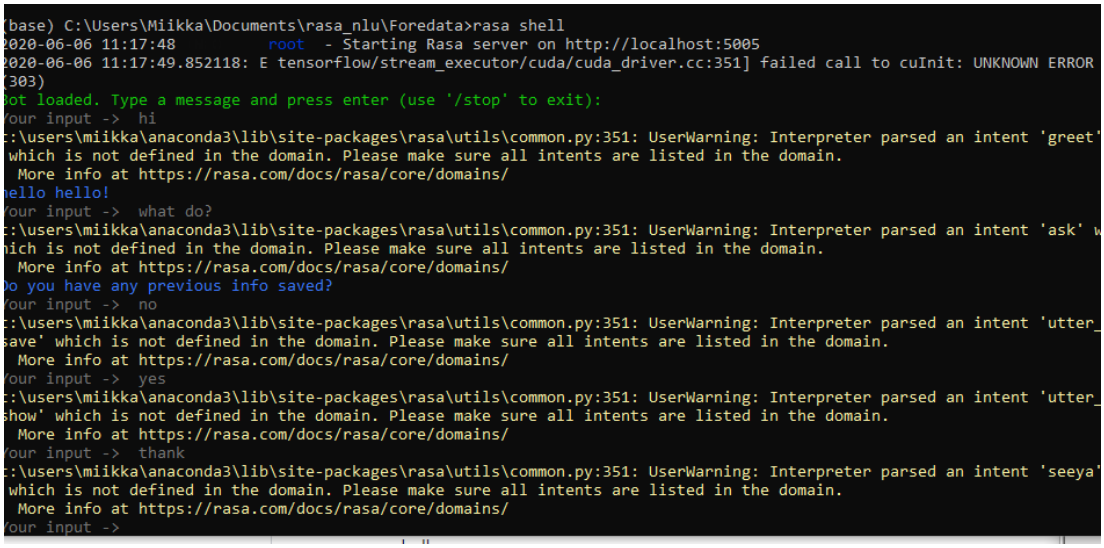
Kuva 26: Nlu.md ja stories.md

Nämä kirjoitettuani laitoin ne Rasan systeemien läpi rasa train komennolla.



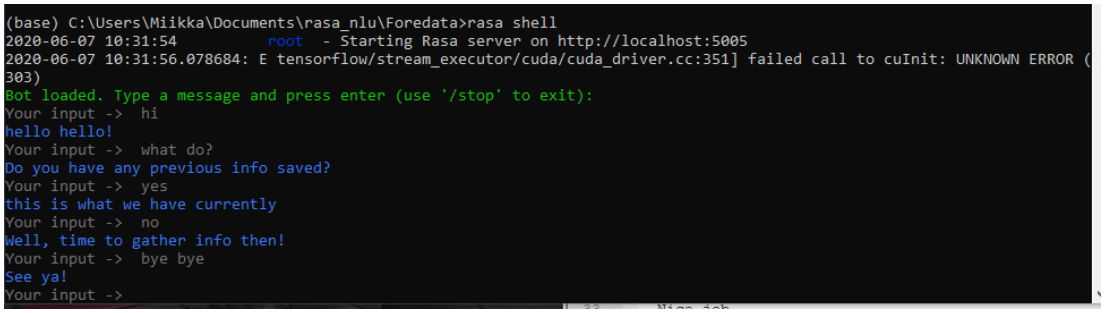
Kuva 27: Rasa train komento

tar.gz on varsinaisen bottini tallennettu muoto, josta rasa ottaa sen käyttöön. Tätä tiedostoa käyttäen sain kuitenkin seuraavan tuloksen.



Kuva 28: Ensimmäinen oma kokeilu.

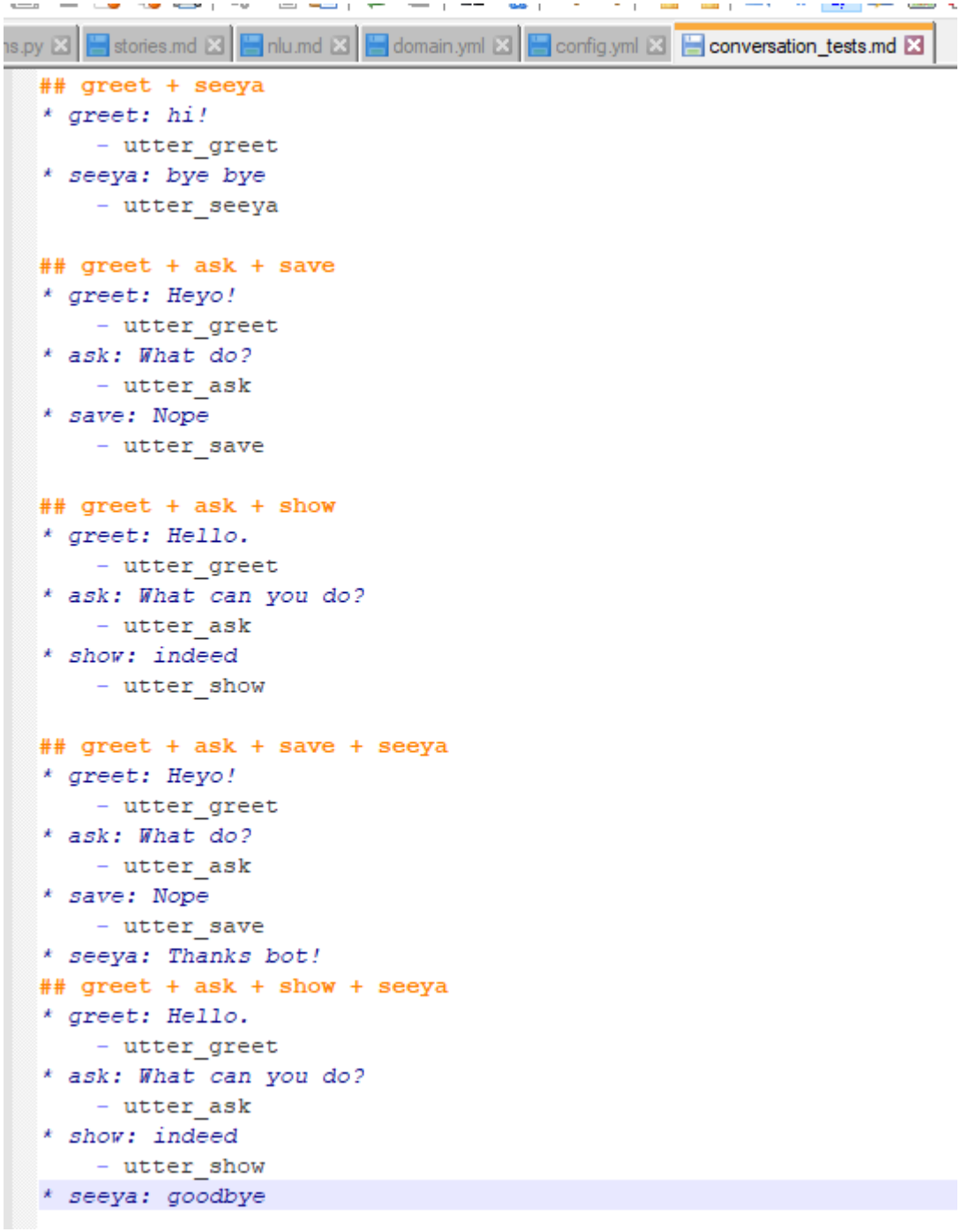
Tässä tilassa botti kyllä löysi meidän intenttimme, mutta se ei ollut domainissa. Lisäksi se ei sanonut mitään, kun sille annettiin no- ja yes- komennot. Antamani koodit olivat uudempia kuin mitä ne tässä tilanteessa olivat. Tärkeimpänä vastauksena tähän paljastui se, että ’vastaukset’ eli ’responses’ ovat eri asia kuin intentit. Ennen korjausta olin kirjoittanut intentit utter_ muodossa, jolloin se ei niitä käyttänyt. Vastausongelma taas johtui siitä, että olin kirjoittanut nimen ‘save’ kahteen kohtaan intenteistä. Botti ei vastaa, jos se on epävarma, kumman sen pitäisi sanoa. Korjausten jälkeen se sanoi seuraavaa rasa shell komennolla:



Kuva 29: Seuraava kokeilu.

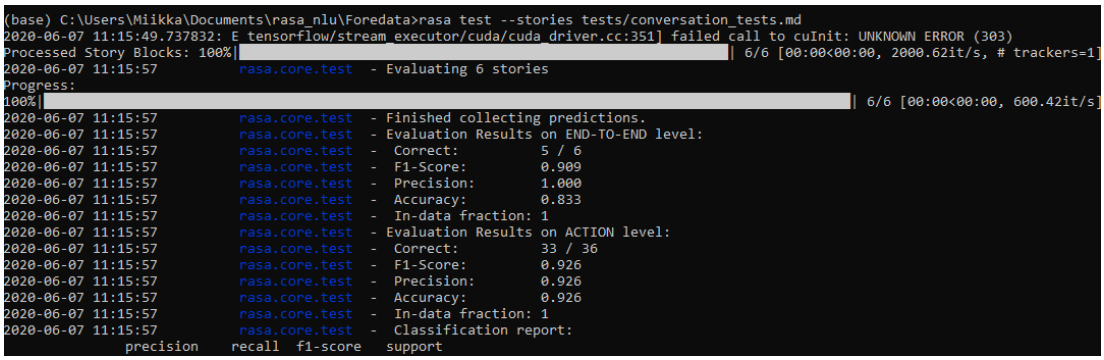
Nyt bottini toimii samalla tavalla kuin esimerkkibotti. Kun sille lähettää viestin, se tarkistaa nlu.md tiedostosta, sisältääkö kommentti mitään niistä viesteistä, ja lähettää sitten vastauksena kyseiseen tekstiin liittyvän vastauksen. Se lähettää myös vastauksen, jos jokin osa lausetta sisältää vastattavan kohteen. Se ei kuitenkaan vastaa, jos se löytää monta vastattavaa vaihtoehtoa. Tätä pyrin korjaamaan seuraavaksi.

Rakennan tests kansion, ja sen sisään laitan conversation_tests.md tiedoston. Tämä antaa botilleni esimerkkikeskusteluja, jotka se ajaa bottimme läpi, ja niiden avulla testaa, toimivatko ne.



Kuva 30: Conversation_tests.md

Tämän jälkeen voin pistää cmd:hen komennon rasa test --stories tests/conversation_tests.md. Lisäsin siihen myös myöhemmin tarkoituksella väärän tarinan, jossa se sanoo greetissä Pöö. Tuloksena on seuraava.



Kuva 31: Testin tulos

Niin kuin kuvasta näkyy, bottini sai kaikki viisi läpi, mutta ei Pöö- tarinaa. Asiat toimivat hyvin. Tietysti näiden testikohteiden määrä saattaa kasvaa paljon suuremmaksi myöhemmin, mutta tällä hetkellä asiat ovat selvät ja yksinkertaiset. Ensin Rasa tuottaa kansioon oman kansionsa ’results’, jonka sisälle se tallentaa failed_stories.md:n. Tämä tiedosto kertoo tarkalleen, mitkä annetuista tarinoista epäonnistuivat.

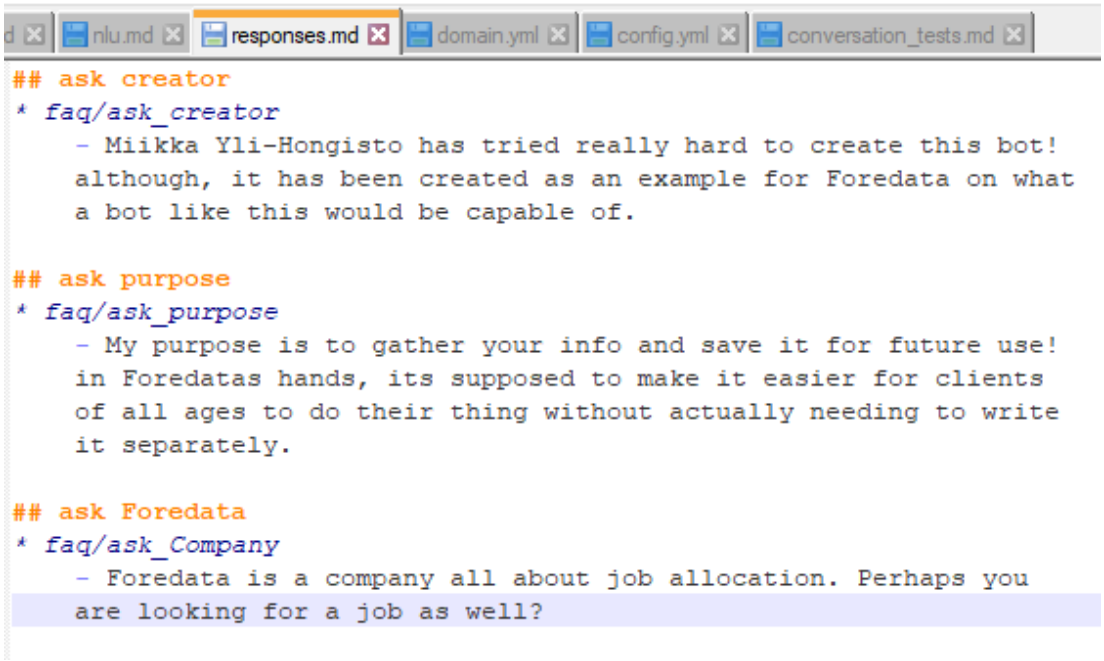
Seuraavassa sivuston kappaleessa käsiteltiin ’response selectoria’. Tämän voi löytää tämänhetkisestä config.yml kappaleesta. Tämä komponentti tekee mahdolliseksi kaikkien tarinoiden laittamisen samaan kansioon. Esimerkkinä tästä rakennan faq kansion, johon laitan mahdollisia kysymyksiä botista, sen tekijästä ja sen omistajasta.

Tätä varten lisään nlu.md tiedostoomme uusia intenttejä.

```
##intent: faq/ask_creator
- Who has created this bot?
- Why are you so well made?
- Where did you come from?
- Who is the owner of this bot?
- Who what and why?

##intent: faq/ask_purpose
- What are you supposed to do?
- I am a bit confused...
- How do you do what you do?
- What is your purpose?

##intent: faq/ask_Foredata
- What is Foredata?
- Why does Foredata want this?
- What is this company about?
- Anything important to tell me about the people behind this?
```

Kuva 32: Uudet intentit

Huomaa, että niissä kaikissa on faq/ osa. Tämä varmistaa sen, että ResponseSelector löytää ja tunnistaa ne faq:n liittyviksi. Lisäksi tein tiedoston ’responses.md’ data kansioomme. Tämä tiedosto sisältää vastaukset näihin kysymyksiin.



Kuva 33: Responses.md

Näiden lisäksi, minun täytyy tietysti lisätä muihin tiedostoihin viittaukset näihin, esimerkiksi domainiin faq intent, ja muutama harjoitus conversation_testsiin.

```
## Some question from faq
* faq
  - respond_faq
```

```
## ask creator
* faq: Who has made this bot?
    - respond_faq
## ask purpose
* faq: I am a bit confused...
    - respond_faq
## ask company
* faq: What is Foredata?
    - respond_faq
```

Kuva 34 ja 35: Lisäykset stories.md:n ja conversations_testsiin.

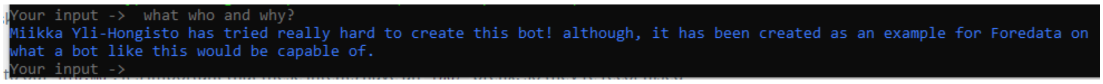
Domain.yml:n lisään kuitenkin myös actions osuuden. Aiemmista kokeiluista tämä voi olla tuttu, koska se oli siellä alusta lähtien.

```
actions:
    - respond_faq
```

Kuva 36: Actions

ResponseSelector vaatii erillisen toiminnon, jotta se voi löytää ne tiedostosta.

Tämän tehtyäni, koska lisäsin uusia tiedostoja bottiin, pistän koodin jälleen rasa trainin läpi, ja sen jälkeen aktivoin rasa shellin, ja kysyn kysymyksen, minkä olen kirjoittanut tarvitsemaan vastauksen faqista. Lopputuloksena saatiin toimiva sovellus.



Kuva 37: Faq kysymys ja vastaus

4.3 Kontekstin lisääminen

Nyt siirryn siihen osioon, jossa alan saada botilleni kontekstin tajua. Tämä tarkoittaa, että saan bottini ymmärtämään, mitä keskustelussa on aiemmin sanottu sen sijaan, että se vain sanoisi sille valmiiksi kirjoitetun vastauksen. Tätä varten teen pienen muutoksen config tiedostoon:

```
policies:
- name: MemoizationPolicy
- name: MappingPolicy
```

Poistimme MemoizationPolicysta MaxHistoryn. Tällä tavalla botti muistaa automaattisesti viimeisimmistä viidestä kommentista tallennettuja asioita, kun se päättää, mitä se sanoo.

Tämän tehtyäni palaan kauan sitten tekemiini suunnitelmiin, ja katson, mitä olioita tarvitsen bottiani varten. Aiempia bottiesimerkkejä kirjoittaessani sain vahvistuksen Foredatalta, että botti ei tule tarvitsemaan erillistä kirjautumista, koska he hoitavat sen toisen järjestelmän kautta. Tämän tiedon vuoksi voin muuttaa valmiin bottini osaamien taitojen listaa kappaleesta 3.1. Valmis bottimme osaa:

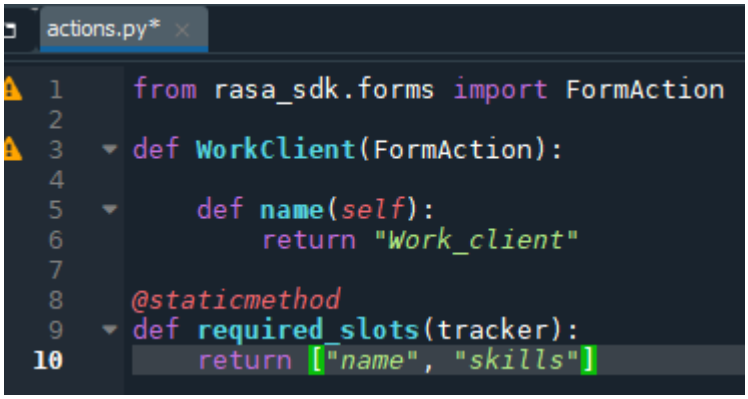
- tervehtiä, kun sitä tervehditään
- pyytää käyttäjän nimen ja taidot tietokantaan tallennusta varten

- näyttää käyttäjän tiedot
- tallentaa kyseiset tiedot
- muuttaa käyttäjän tietoja pyynnöstä
- tarjota työpaikkaa perustuen käyttäjän taitoihin.

Voin myös muuttaa aikomuslistaa samalla tavalla.

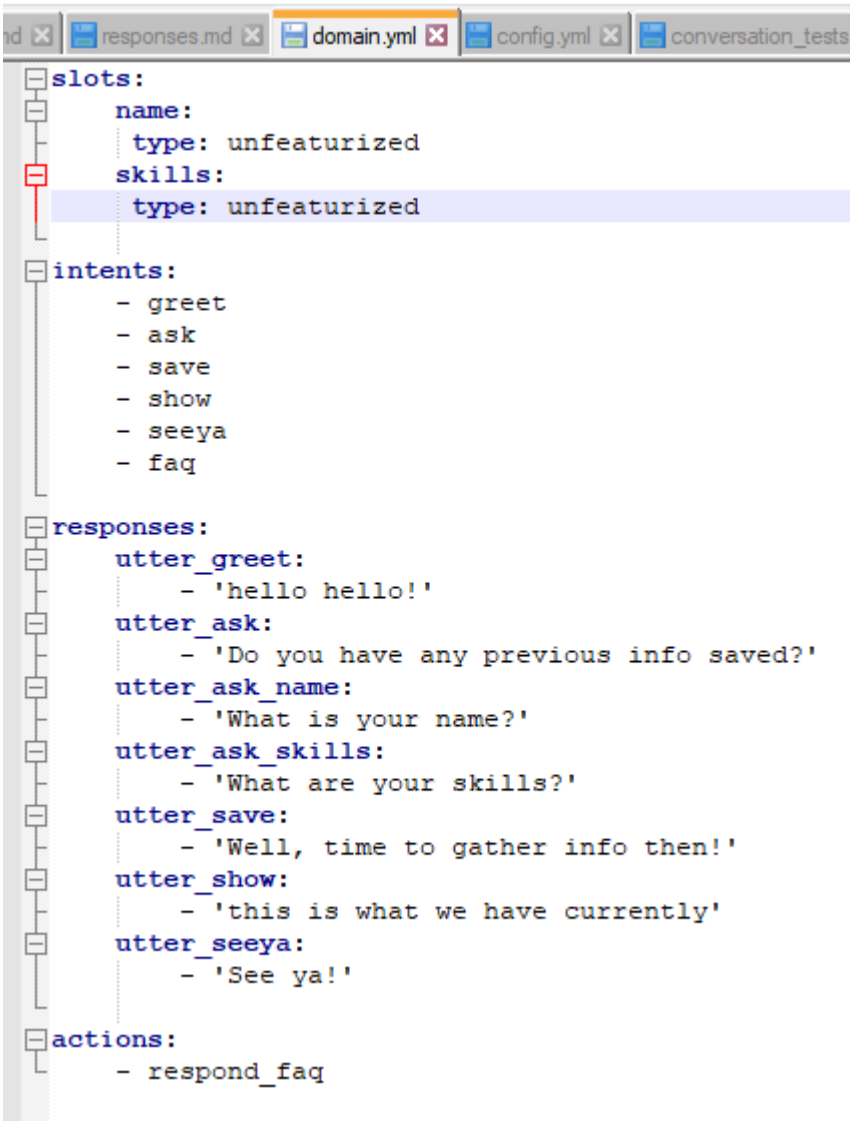
- käyttäjä tervehtii chattibottia
- käyttäjä haluaa tehdä itselleen käyttäjän sivustolle
- käyttäjä haluaa katsoa tietojaan
- käyttäjä haluaa muuttaa tietojaan
- käyttäjä haluaa työpaikkaehdotuksen.

Tällä tavalla voin tulla siihen tulokseen, että tarvitsen pääasiassa kaksi oliota bottia varten: Kohteen nimi ja työtaidot. Tulevaisuudessa tulen myös tarvitsemaan jonkinlaisen olion, johon voin säilöä työpaikkaehdotuksia, mutta tällä hetkellä keskityn vain näihin kahteen.



Kuva 38: Actions.py

Rakennan actions.py tiedostoon seuraavat tiedot. Koodin avulla annan botille nimen ”Work_client”, jolla voin kutsua sitä myöhemmin bottia varten. Lisäksi rakensin required_slots metodin, joka kertoo sille, mitä olioita sen täytyy käyttää.
Tietysti, jotta se pystyy niitä käyttämään, minun tarvitsee rakentaa tämänhetkiseen tiedostooni paikat niille. Tätä varten lisään ”slots” kohdan domain-tiedostoon, ja myös uusia response kohtia, jossa bottimme pyytää näitä tietoja.



Kuva 39: Domain päivitys

Tämän jälkeen lisään myös actions.py:hyn kohdan ’submit’.

```
def submit(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict]:

    dispatcher.utter_message("Thanks for getting in touch, we'll contact you soon")
    return []
```

Kuva 40: Def submit

Tällä hetkellä tämä koodi sanoo vain saaneensa tiedot ja kiittää siitä. Sen tarkoitus on yleensä kuitenkin tallentaa ne jonkinlaiseen tietokantaan. Ohjeen mukaan tämä toteutetaan myöhemmin.

Seuraavaksi lisätään jälleen uusi osa domain-tiedostoon. Tällä kertaa forms, ja sen sisälle work_client. Lisäksi teen intentteihin kohdan, jossa kutsun tätä koodia, ja nlu.md:hen kohdan, joka tunnistaa, mitä se kerää ihmisten kommenteista. Storiesiin lisään kohdan, joka aktivoi koodin, ja policieseihin laitan vielä kaksi uutta policya, jotta botti toimii halutulla tavalla: KerasPolicy ja FormsPolicy. Kaikkiin lisätään viittaus näihin uusiin asioihin.

```
policies:
- name: MemoizationPolicy
- name: MappingPolicy
- name: KerasPolicy
- name: FormPolicy
```

Kuva 41: Policies

```
##intent: inform
- my name is [Maria paladin](name)
- [matti meikäläinen](name)
- I am named [Kalle Harjuvaara](name)
- [Lars Legend](name)
- [Jane Doe](name)
- [John Doe](name)
- my skills are [cooking, baking and cleaning]. (skills)
- [driving a truck, construction and guard duty]. (skills)
- I can do [traffic duty, engineering and programming]. (skills)
- [Teaching, group projects and taking care of children]. (skills)
```

Kuva 42: Intent: inform

```
## info getting
* gather_info
  - work_client                  <!--run the work_client action-->
  - form{"name": "work_client"} <!--activate the form-->
  - form{"name": "null"}         <!--Deactivate the form-->
```

Kuva 43: Stories.md:n kohta work_clientille

```yaml
slots:
    name:
     type: unfeaturized
    skills:
     type: unfeaturized

intents:
    - greet
    - ask
    - save
    - show
    - seeya
    - gather_info
    - inform
    - faq

entities:
    - name
    - skills

responses:
    utter_greet:
        - 'hello hello!'
    utter_ask:
        - 'Do you have any previous info saved?'
    utter_ask_name:
        - 'What is your name? Give both your first and surname.'
    utter_ask_skills:
        - 'What are your skills? give all of them at once.'
    utter_save:
        - 'Well, time to gather info then!'
    utter_show:
        - 'this is what we have currently'
    utter_seeya:
        - 'See ya!'

forms:
    - work_client

actions:
    - respond_faq
```

Kuva 44: Domain.yml nyt

Kaiken tämän lisäksi, teen vielä yhden uuden tiedoston kansioon: endpoints.yml. Tämä koodi tulee päättämään, missä kohtaa se lopettaa keskustelun.

```yaml
action_endpoint:
  url: "http://localhost:5055/webhook"
```

Kuva 45: Endpoints.yml

Nyt voin käynnistää botin uudestaan. Mutta sitä ennen, tehdään actions serveri. Bottimme tulee käyttämään sitä, jotta se voi käyttää sinne ohjelmoituja toimintoja.

```
2020-06-12 10:44:50          rasa_sdk.endpoint  - Starting action endpoint server...
2020-06-12 10:44:50          rasa_sdk.executor  - Registered function for 'work_client'.
```

Kuva 46: Actions toimii

Tämän jälkeen kuitenkin ongelmaksi nousi uusi asia. Tällä kertaa vaikuttaa siltä, että minun pitäisi tehdä required_slots kohta koodiin, mutta sellainen on jo molempien kohtien kanssa, niin kuin kuvasta 38 näkyy. Kysymyksenä kuuluu siis, mikä on koodissa ongelmana? Botti löytää work_clientin, mutta jostain syystä ei yhdistä sitä siihen.

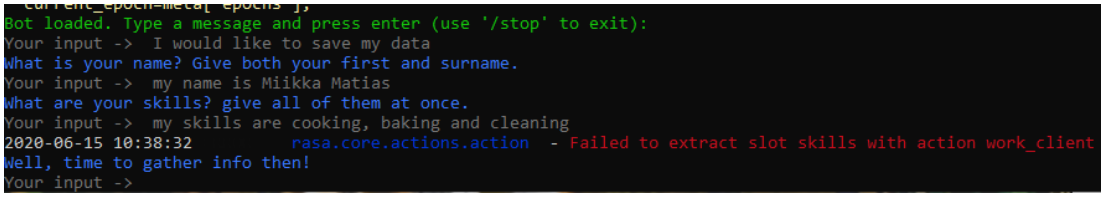
```
  File "C:\Users\Miikka\AppData\Roaming\Python\Python37\site-packages\rasa_sdk\forms.py", line 35, in required_slots
    "A form must implement required slots that it has to fill"
NotImplementedError: A form must implement required slots that it has to fill
```

Kuva 47: Notimplementederror

Syynä virheeseen oli sisennyksiä käyttävä konfiguraation syntaksi. Botti ottaa vastaan nimen melko valikoivasti, joskus ottaen vain tarkan esimerkin, mutta työn kanssa sillä on enemmän ongelmia. Jostain syystä se myös antaa aina utter_save kohdan lausahduksen, kun se menee pieleen. Se on teoriassa seuraava kohta listalla, joten en ole varma, onko sen tarkoitus toimia näin.

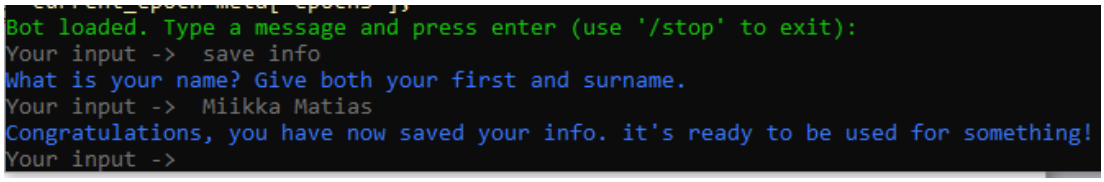
```python
from rasa_sdk.forms import FormAction
from typing import Any, Text, Dict, List

from rasa_sdk import Tracker
from rasa_sdk.executor import CollectingDispatcher

class WorkClient(FormAction):

    def name(self):
        return "work_client"

    @staticmethod
    def required_slots(tracker):
        return ["name", "skills"]

    def submit(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict]:

        dispatcher.utter_message("Congratulations, you have now saved your info. it's ready to be used for something!")
        return []
```

Kuva 48: Siistimpi actions.py



Kuva 49: Jotenkin toimii

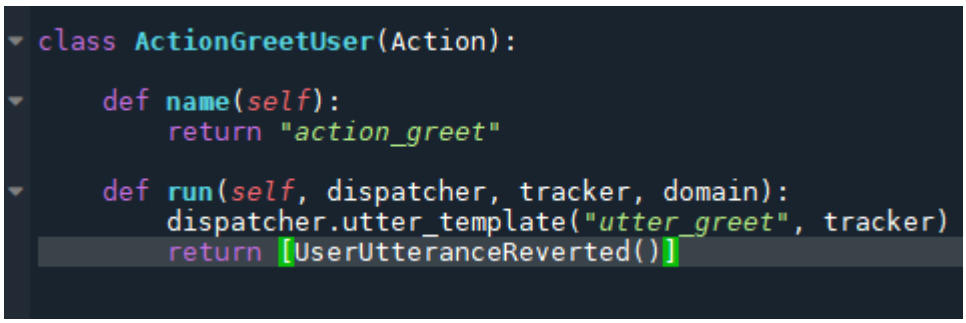
Päätin poistaa skills-kohdat listoista ja hoidan asiaa enemmän myöhemmässä vaiheessa. Tämän tehtyäni botti toimii odotetusti, mutta se tarkoittaa, että joudumme tulevaisuudessa ottamaan selville, miksi bottimme ei löydä skills-kohtaa.



Kuva 50: Yksinkertaistettu versio.

Nyt kun botti tekee jotain konkreettista, voidaan alkaa hoitaa odottamattomia kommentteja. Tällainen voi esimerkiksi olla tervehdys siinä vaiheessa, kun botti pyytää käyttäjää antamaan nimensä. Tietysti botti saattaa yrittää tallentaa tervehdyksen nimenä, mutta voimme estää sen tässä tilanteessa. Emme pysty tietenkään tunnistamaan kaikkia tällaisia tapauksia, mutta helpoimmat voimme estää.

Tätä varten teen actions.py:hyn uuden kohdan ‘ActionGreetUser’. Lisäksi muutan greet intenttiä niin, että se aktivoi kyseisen koodin, ja poistan greetin stories.md:stä.



Kuva 51: ActionGreetUser

```
- greet: {triggers: action_greet}
```

Kuva 52: Intent greet

Tämän jälkeen voin jälleen testata bottia. Niin kuin kuvasta 53 näkyy, bottimme pystyy vieläkin tervehtimään, mutta nyt tervehtiminen ei välttämättä keskeytä, mitä henkilö oli tekemässä.

```
  current_epoch=meta["epochs"],
Bot loaded. Type a message and press enter (use '/stop' to exit):
Your input ->  hi
hello hello!
Your input ->  save info
What is your name? Give both your first and surname.
Your input ->  hi
hello hello!
Your input ->
```

Kuva 53: Toimii

Voimme tehdä saman myös faq-kysymyksille. Tätä varten en kuitenkaan tarvitse uutta kohtaa actionsiin, vaan voin myös pistää stories.md:hin erillisen kohdan sitä hoitamaan.

```
## Name and continue
* contact_work
    - work_client
    - form{"name": "work_client"}
* faq
    - respond_faq
    - work_client
    - form{"name": null}
```

Kuva 54: Stories Name and continue

Tämän pitäisi johtaa siihen, että botti jatkaa tiedon keräämistä, vaikka se kysyisikin apua välissä. Kun yritin sitä käyttää tosin, bottimme toimi vähän erikoisesti:

```
Your input ->  save data
What is your name? Give both your first and surname.
Your input ->  i am a bit confused...
2020-06-17 11:12:13          rasa.core.actions.action  - Failed to extract slot name with action work_client
My purpose is to gather your info and save it for future use! in Foredatas hands, its supposed to make it easier for clients
 of all ages to do their thing without actually needing to write it separately.
What is your name? Give both your first and surname.
Your input ->  Miikka matias
Congratulations, you have now saved your info. it's ready to be used for something!
Your input ->
```

Kuva 55: Toimii kai?

Tämän lisäksi pystyn myös tekemään vastauksia kysymyksiin tietyistä parametreista. Tätä varten lisään slotsiin kohdan ‘requested slot’.

```
slots:
    requested_slot:
        type: categorical
        values:
            - name
```

Kuva 56: Requested slot

Tämän avulla koodi tunnistaa erikseen mistä kohdasta puhutaan, kun laitan sen samanlaiseen tarinaan kuin name and continue:

```
## explain name
* gather_info
    - work_client
    - form{"name": "work_client"}
    - slot{"requested_slot": "name"}
* explain
    - utter_explain_why_name
    - work_client
    - form{"name": null}
```

Kuva 57: Explain name

Näiden kahden lisäksi tietysti lisään muihin osioihin myös explain osion. Tämän jälkeen voin ajaa jälleen tiedoston. Aiempi ongelma siitä, että botti valittaa, ettei saanut kohtaa täytettyä, on vielä täällä, mutta muuten se toimii:

```
Your input ->  save info
What is your name? Give both your first and surname.
Your input ->  why?
2020-06-22 10:41:52          rasa.core.actions.action  - Failed to extract slot name with action work_client
Well, we cannot recognize you without your name after all!
What is your name? Give both your first and surname.
Your input ->
```

Kuva 58: Toimii jälleen.

Viimeisenä asiana tämä tutoriaali opettaa, miten saan botin myöntämään, että se ei ymmärtänyt jotain. Jos minä rakennan bottini käyttämään myöhemmin suomen kieltä, tämä tulee olemaan erityisen tärkeätä eri sanamuotojen kanssa. Tätä varten lisään

config tiedostoon FallbackPolicies kohdan ja lisään intenteihin ‘out_of_scope’ intentin:

```
- name: TwoStageFallbackPolicy
  nlu_threshold: 0.8
```

Kuva 59: Fallbackpolicy

Luku 0.8 tarkoittaa tässä tilanteessa, kuinka hyvin botin tarvitsee ymmärtää annettu kysymys, ennen kuin se lähettää tämän viestin. Tätä lukiessanne olette saattaneet nähdä botin harjoittaessa sen nostavan esille prosentteja. Jos prosentit ovat alle 0.8, botti lähettää viestin, että se ei ymmärtänyt asiaa. Tämä viesti tarvitsee tietysti koodata. Lisään domainin responses kohtaan Out_of_scope kohdan, ja teemme saman stories tiedostoon. Loppujen lopuksi voin jälleen ajaa tiedoston läpi.

```
Your input ->  who is the US president?
That is not part of my job description
Your input ->  what am I doing here?
My purpose is to gather your info and save it for future use! in Foredatas hands, its supposed to make it easier for cli
ents of all ages to do their thing without actually needing to write it separately.
Your input ->  do you serve food?
? Did you mean 'out_of_scope'?  1: Yes (/out_of_scope)
? Did you mean 'out_of_scope'?  1: Yes (/out_of_scope)
That is not part of my job description
Your input ->
```

Kuva 60: Kysymys ja toiminta

Vaikuttaa siltä, että Fallbackpolicy on myös lisännyt tilanteen, jossa botti kysyy erikseen, mitä tarkoitamme ongelman noustessa. Tämä kiinnittää huomiota, mutta botti vaikuttaa toimivan niin kuin sen pitäisi siitä huolimatta.

Kokemuksena botin rakentaminen oli paljon mielekkäämpää verrattuna kirjan esimerkkeihin. Varsinaisen botin kanssa keskustelu teki työstä paljon viihdyttävämpää, ja antoi paljon paremman kuvan siitä, että työni etenee. Seuraava askel on kuitenkin se, että alan rakentamaan botilleni enemmän omaa kuin lainattua koodia. Suurin osa muutoksista, joita olen tähän mennessä tehnyt koodiin, on ollut vain personalisointia, ja bottimme toimii suurin piirtein samalla tavalla kuin sivulla.

5 FOREDATAN OHJEIDEN SEURANTA JA ONGELMIEN KORJAUS

5.1 Mitä ongelmia toteutuksessa oli?

Tässä vaiheessa voin alkaa rakentamaan koodiani vielä yksilöllisemmäksi Foredatan pyyntöjen mukaan. Ensimmäiseksi kuitenkin ryhdyn korjaamaan joitakin ongelmia, tai selvimmin puuttuvia osia. Esimerkiksi aiemmin mainittu skills-kohta ja kyseisten taitojen tallennus ja uusi ongelma, jossa bottimme vaikuttaa lukittautuvan ottamaan vastauksia vastaan. Keskityn ensimmäiseksi näiden ongelmien korjaamiseen.

Skills-kohdan ongelmana oli aiemmin, että se ei ottanut vastaan mitä sille annettiin, eikä pystynyt tallentamaan tietoja. Samaa ongelmaa ei ollut name-kohdan kanssa, vaikka ne oltiin tehty samalla tavalla koodiin. Päätin kokeilla mitä tapahtuu, jos lisään skills-kohdat takaisin koodiin kaikkialle, missä name oli myös osana. Lisäksi tein sille myös muutaman muun kohdan, jotka olemme lisänneet sen jälkeen, kun sen poistimme.

```
## skills getting
* gather_info
  - work_client
  - form{"skills": "work_client"}
  - form{"skills": null}
```

Kuva 61: Gather skills

```
## explain skills
* gather_info
    - work_client
    - form{"name": "work_client"}
    - slot{"requested_slot": "skills"}
*explain
    - utter_explain_why_skills
    - work_client
    - form{"name": null}
```

Kuva 62: Explain skills

```
## Skills and continue
* gather_info
    - work_client
    - form{"skills": "work_client"}
* faq
    - respond_faq
    - work_client
    - form{"name": null}
```

Kuva 63: Skills and continue

Metodi gather_info kerää skillsiä pyytäessä sen tiedot, explain skills kertoo kysyttäessä, miksi me sitä pyydämme, ja ‘Skills and continue’ antaa asiakkaan kysyä faq -kysymyksiä ilman, että se keskeyttää pyyntöjen täyttöprosessia.

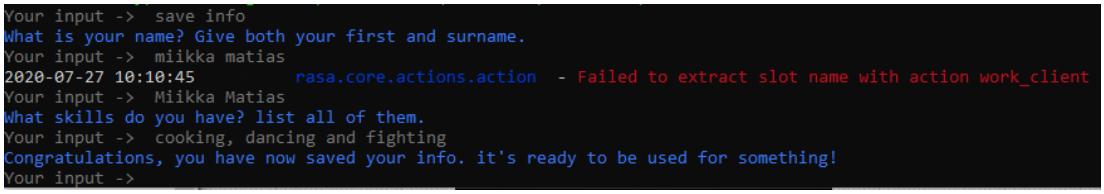
Tämän jälkeen voin jälleen kokeilla, miten botti toimii skillsien kanssa.

```
Bot loaded. Type a message and press enter (use '/stop' to exit):
Your input ->  save info
What is your name? Give both your first and surname.
Your input ->  Miikka Matias
What skills do you have? list all of them.
Your input ->  why?
2020-07-06 11:05:41         rasa.core.actions.action  - Failed to extract slot skills with action work_client
Your skills are what we will send to your to be employers
What skills do you have? list all of them.
Your input ->
```

Kuva 64: skills kysymys

Muuten toiminto toimii normaalisti, mutta se nostaa esille oletetun vastauksen, ja sitten siirtyy takaisin kysymykseen. Ongelma vaikuttaa olevan work_clientissa, ja miten koodi on siellä kirjoitettu.

Otettuani yhteyttä Foredataan ja esitellessäni työni tuloksia heille, huomasin kuitenkin mielenkiintoisen tilanteen: kun kirjoitin nimeni pienellä, bottimme ei sitä suostunut tallentamaan, mutta kun annoin niille isot alkukirjaimet, se teki sen odotetusti. Lisäksi, kun kirjoitin työtaitoni suoraan nlu.md tiedostoomme kirjoittaman tekstin mukaan, botti tallensi senkin normaalisti.



Tarkasteltuani nlu.md tiedostoa tarkemmin, huomasinkin nimen suhteen, että kaikki esimerkkini sitä varten oli kirjoitettu isolla. Tämän takia, botti hyväksyy vain isolla alkukirjaimella alkavat nimet, mutta se hyväksyy myös ne, jotka eivät ole listalla. Botti on selvästi oppinut tavan, jolla käyttäytyä kirjoittamani koodin mukaan. Ongelma ei siis ole koodissa, vaan millä tavalla botti on opetettu ottamaan vastaan tietoja. Tätä varten tarvitsen joko enemmän esimerkkejä, joihin verrata, tai sitten voin ohjelmoida botin erityisesti pyytämään ne jossain tietyssä muodossa.

Seuraava asia mitä yritän, on saada resetoitua keskustelu takaisin alkuun ilman, että bottimme tarvitsee käynnistää uudestaan. Niin kuin esitin aiemmassa kuvassa, tietojen pyynnön jälkeen bottimme alkoi vastaamaan kaikkeen kommentilla ‘Congratulations, you have now saved your info.’ Tätä varten löysin korjaustavan helposti. Minun tarvitsi vain lisätä actions.py tiedostoon rasa sdk.events kirjastosta toiminto Restarted. Tämän jälkeen lisäsin sen botissa muuten tyhjänä olleeseen returniin.

```
from rasa_sdk.forms import FormAction
from typing import Any, Text, Dict, List

from rasa_sdk import Tracker
from rasa_sdk.executor import CollectingDispatcher
from rasa_sdk import Action
from rasa_sdk.events import UserUtteranceReverted

from rasa_core_sdk.events import Restarted


class WorkClient(FormAction):

    def name(self):
        return "work_client"

    @staticmethod
    def required_slots(tracker):
        return ["name",
                "skills",]

    def submit(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict]:

        dispatcher.utter_message("Congratulations, you have now
        return [Restarted()]

class ActionGreetUser(Action):

    def name(self):
        return "action_greet"

    def run(self, dispatcher, tracker, domain):
        dispatcher.utter_template("utter_greet", tracker)
        return [UserUtteranceReverted()]
```

Kuva 66: Restarted lisäykset

Tämä korjasi ongelmani niin, että bottimme teoriassa menee takaisin botin alkuun ilman, että se menettää vastaanottamiaan tietoja.

5.2 Verkkoon tallennus

Seuraavaksi kerätyt tiedot pitää tallentaa verkkoon. Tämä on ollut iso keskustelunaihe Foredatan kanssa, koska emme kumpikaan olleet varmoja siitä, mihin me nämä tiedot tallentaisimme. Löysin kuitenkin verkosta esimerkin, miltä kyseisen koodin pitäisi näyttää, jos tiedot tallennetaan google driveen.

Ensiksi kuitenkin päätin kokeilla, miten koodini yleisesti toimii. Google drivea varten ohjelman tarvitsee ensin pyytää lupa, jotta se suostuu antamaan siellä olevat tiedot botin käyttöön ja yleisesti muokata niitä. Tätä varten lataan tarvittavat kirjastot verkosta seuraavalla komennolla:

```
pip3 install google-api-python-client google-auth-httplib2 google-auth-oauthlib tabulate requests tqdm
```

Kuva 68: Tabulate

Tämän tehtyäni, siirryn sivustolle https://developers.google.com/drive/api/v3/quickstart/python ja suoritan toiminnon ’Enable the drive api’. Tämä antaa minun ladata verkosta itselleni credentials.json tiedoston, jonka tallennan Foredata-kansioon tulevia osia varten.

Tämän jälkeen rakennan esimerkkinä botin toiminnasta quickstart.py tiedoston. Tämä on aika paljon suurempi kuin actions tiedostomme, joten olen jakanut sen kolmeen eri kuvaan, joissa olevista luokista kerron enemmän jokaisen kuvan alla.

```python
import pickle
import os
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from tabulate import tabulate

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/drive.metadata.readonly']

def get_gdrive_service():
    creds = None
    # The file token.pickle stores the user's access and refresh tokens, and is
    # created automatically when the authorization flow completes for the first
    # time.
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)
    # If there are no (valid) credentials available, let the user log in.
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open('token.pickle', 'wb') as token:
            pickle.dump(creds, token)
    # return Google Drive API service
    return build('drive', 'v3', credentials=creds)

def get_size_format(b, factor=1024, suffix="B"):
    """
    Scale bytes to its proper byte format
    e.g:
        1253656 => '1.20MB'
        1253656678 => '1.17GB'
    """
    for unit in ["", "K", "M", "G", "T", "P", "E", "Z"]:
        if b < factor:
            return f"{b:.2f}{unit}{suffix}"
        b /= factor
    return f"{b:.2f}Y{suffix}"
```

Kuva 69: Quickstart.py 1

Get_gdrive_service on virallisesti se, joka antaa koodille oikeuden käsitellä ja nähdä google drivessa olevat tiedostot. Tätä varten se tekee ’token.pickle’ tiedoston kansioon, mitä koodi käyttää varmistamaan, että sillä on oikeus tiedostoon. Tämä tiedosto pitää aina poistaa, jos koodiin lisätään joku uusi toiminto, koska uudet oikeudet pitää aina pyytää erikseen.

Get_sizeformat on yksinkertainen funktio, joka kertoo kuinka iso tietty tiedosto google drivessa on.

```python
def list_files(items):
    """given items returned by Google Drive API, prints them in a tabular way"""
    if not items:
        # empty drive
        print('No files found.')
    else:
        rows = []
        for item in items:
            # get the File ID
            id = item["id"]
            # get the name of file
            name = item["name"]
            try:
                # parent directory ID
                parents = item["parents"]
            except:
                # has no parrents
                parents = "N/A"
            try:
                # get the size in nice bytes format (KB, MB, etc.)
                size = get_size_format(int(item["size"]))
            except:
                # not a file, may be a folder
                size = "N/A"
            # get the Google Drive type of file
            mime_type = item["mimeType"]
            # get last modified date time
            modified_time = item["modifiedTime"]
            # append everything to the list
            rows.append((id, name, parents, size, mime_type, modified_time))
        print("Files:")
        # convert to a human readable table
        table = tabulate(rows, headers=["ID", "Name", "Parents", "Size", "Type", "Modified Time"])
        # print the table
        print(table)
```

Kuva 70: Quickstart.py 2

List_files on myös yksinkertainen funktio joka listaa löytämiensä tietojen osat pöytä muodossa (kuva 73). Tämä on se kohta, jossa käytetään tabulatea, minkä latasin aiemmin.

```python
def main():
    """Shows basic usage of the Drive v3 API.
    Prints the names and ids of the first 5 files the user has access to.
    """
    service = get_gdrive_service()
    # Call the Drive v3 API
    results = service.files().list(
        pageSize=5, fields="nextPageToken, files(id, name, mimeType, size, parents, modifiedTime)").execu
    # get the results
    items = results.get('files', [])
    # list all 20 files & folders
    list_files(items)

if __name__ == '__main__':
    main()
```

Kuva 71: Quickstart.py 3

Main käynnistää ja tulostaa aiempien osien kohdat.

```python
def upload_files():
    """
    Creates a folder and upload a file to it
    """
    # authenticate account
    service = get_gdrive_service()
    # folder details we want to make
    folder_metadata = {
        "name": "TestFolder",
        "mimeType": "application/vnd.google-apps.folder"
    }
    # create the folder
    file = service.files().create(body=folder_metadata, fields="id").execute()
    # get the folder id
    folder_id = file.get("id")
    print("Folder ID:", folder_id)
    # upload a file text file
    # first, define file metadata, such as the name and the parent folder ID
    file_metadata = {
        "name": "test.txt",
        "parents": [folder_id]
    }
    # upload
    media = MediaFileUpload("test.txt", resumable=True)
    file = service.files().create(body=file_metadata, media_body=media, fields='id').execute()
    print("File created, id:", file.get("id"))
```

Kuva 72: Upload_files

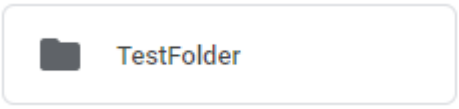
Tämä koodi on omaa toimintaani varten tärkein. Se tekee google driveeni kansion, ja siirtää sinne koneelta tiedoston test.txt. Testitiedoston tarvitsee kuitenkin sisältää jotain, tai muuten se osoittaa virhettä, mutta tämä ei olisi ongelma botin kanssa, koska nimet ja työtaidot kulkisivat botin kysymysten läpi. Tätä varten kuitenkin lisäämme koodiin samanlaisen kohdan kuin Mainille, joka käynnistää sen.

Tämän jälkeen voin siirtyä cmd:hen ja käynnistää sen pythonin kautta komennolla ’python quickstart.py’. Tämä avaa selaimeni, ja pyytää minua sieltä kirjautumaan sisään googleen. Tämän jälkeen, koodi pyytää oikeutta käyttää tiedostojamme ja lisätä niitä sinne. Hyväksyttyäni ne, sivusto vie meidät erilliselle sivustolle, joka kertoo meille toiminnon onnistuneen ja katsoessamme cmd:hen, voimme nähdä seuraavaa:

```
(base) C:\Users\Miikka\Documents\rasa_nlu\Foredata>python quickstart.py
Files:
ID                                          Name                                                Parents                                 Size    Type                                    Modi
fied Time
--------------------------------------------  --------------------------------------------------  --------------------------------------  ------  --------------------------------------  ----
-----------------
1RD0etqebVW7DCC_jtv952JWr0hOPvy4ZZxTNhQ9aWE4  Legends of Runeterra Expedition Tier List by KrabKore  N/A                                     N/A     application/vnd.google-apps.spreadsheet  2020
-08-13T06:14:53.807Z
1Ne_B2ZnOXrtu5m87x8NTcB3cj2f1rvz0            test.txt                                            ['1vbBi8FG0asFHhuzhdZjoInNqU7WtGh1f']   10.00B  text/plain                              2020
-08-12T12:18:09.354Z
1vbBi8FG0asFHhuzhdZjoInNqU7WtGh1f            TestFolder                                          ['0AJvgBY_gEGHvUk9PVA']                 N/A     application/vnd.google-apps.folder       2020
-08-12T12:18:08.174Z
1Ur3YeTR-8Kuo5-V7bKiYM3gPW849I-xd            TestFolder                                          ['0AJvgBY_gEGHvUk9PVA']                 N/A     application/vnd.google-apps.folder       2020
-08-12T12:11:39.106Z
1zW0eulkvUi45ih3K_C-iFS4D-uqwwRLU            TestFolder                                          ['0AJvgBY_gEGHvUk9PVA']                 N/A     application/vnd.google-apps.folder       2020
-08-11T07:45:24.350Z
Folder ID: 1_NaSnp_nmWg2OvWdg2q0y39sg82CMWcP
File created, id: 1tS57qFUp150XPpeJXpPM-O8C2aZ_F0_S
```

Kuva 73: Quickstart lopputulos.

Niin kuin tekstistä voi nähdä, testikansio on onnistuttu luomaan. Koodi listaa viisi viimeisintä tiedostoa, jotka ovat google drivessa olleet, ja antaa folder id:n. Lisäksi se kertoo meille lähettämänsä tiedoston id:n.



Kuva 74: Testikansio.

Tämän jälkeen olisi ideana jotenkin yhdistää edellinen omaan bottiini, jotta se laittaisi keräämämme tiedot samalla tavalla kansioon. Tässä tilanteessa otaksun, että ohjelma kysyisi samalla tavalla oikeutta käyttää google drivea ensimmäisellä käynnistyskerralla kuin quickstartin kanssa. Mutta koska actions.py tekee toimivan palvelimen bottimme toiminnoille, en ole varma, toimisiko se samalla tavalla.

Alkuun päätin rohkeasti vain kopioida ’get_gdrive_service’ ja ’upload_files’ kohdat koodista. En lisää muita, ennen kuin tiedän että tämä toimii itsessään. Rasa run actions komennon päälle laittaessa se ottaa vain class funktiot käyttöön, mutta laitettuani komennon actions.py, se vei minut samanlaisen konfiguraatio prosessin läpi kuin aiemmin.

Tämän jälkeen tein yksinkertaisen tekstitiedoston nimeltä tiedot.txt, ja koodasin botin kirjoittamaan sen keräämät tiedot sinne tiedonkeräyksen jälkeen seuraavalla koodilla:

```
    name = tracker.get_slot("name")
    skills = tracker.get_slot("skills")

    with open("tiedot.txt", "a") as f:
        f.write("\n")
        f.write("\n")
        f.write(name)
        f.write("\n")
        f.write(skills)

    main()
```

Kuva 75: Txt tiedostoon kirjoitus

Kuvassa \n kirjoitukset tarkoittavat, että teksti vaihtaa riviä automaattisesti, ja kirjoittaa tekstin siihen. Tämä varmistaa sen, että tiedot eivät jatku epäsiistissä jonossa, vaan jokaisen nimen alla on tietyn asiakkaan omat työtaidot. Kuvassa oleva main-osuus taas ottaa käyttöön aiemmin kirjoittamani koodin, jonka olen laittanut oman

koodini muotoon. Koodin tuloksena, tiedot.txt tiedostoon ilmestyy seuraava teksti kahden botin käyttäjän jälkeen.

```
Lars Legend
cooking, dancing and fighting
Jane Doe
driving, patting, caretaking
```

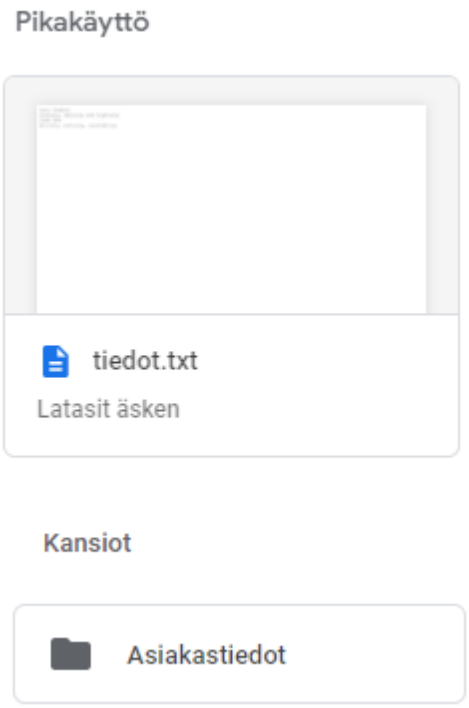
Kuva 76: Tiedot.txt

Main-kohta taas näyttää bottia varten tältä:

```python
def main():
    """Shows basic usage of the Drive v3 API.
    Prints the names and ids of the first 5 files the user has access to.
    """
    service = get_gdrive_service()
    folder_metadata = {
        "name": "Asiakastiedot",
        "mimeType": "application/vnd.google-apps.folder"
    }
    # create the folder
    file = service.files().create(body=folder_metadata, fields="id").execute()
    # get the folder id
    folder_id = file.get("id")
    print("Folder ID:", folder_id)
    # upload a file text file
    # first, define file metadata, such as the name and the parent folder ID
    file_metadata = {
        "name": "tiedot.txt",
        "parents": [folder_id]
    }
    # upload
    # Ei tallenna tyhjiä tiedostoja
    media = MediaFileUpload("tiedot.txt", resumable=True)
    file = service.files().create(body=file_metadata, media_body=media, fields='id').execute()
    print("File created, id:", file.get("id"))
```

Kuva 77: Def main()

Botti toteuttaa tämän koodin luoden google driveen kansion ’asiakastiedot’ ja tallentaa tekstitiedoston sen sisään. Tuloksena saan seuraavan:



Kuva 78: Botin tulos

Nyt botti siis kerää tiedot, tallentaa ne verkkoon txt-muodossa ja voi vastata asiakkaan kysymyksiin tarvittaessa. Mielestäni botti on käyttökelpoinen, ja tästä eteenpäin sitä on helppoa hioa siihen suuntaan, mihin Foredata itse haluaa sen viedä.

6 YHTEENVETO JA PÄÄTELMÄT

Chattibotit työnä ovat osoittautuneet paljon laajemmiksi kohteiksi kuin oletin. Se, että ne ovat vieläkin selvästi kehitysvaiheissa, on hyvin mielenkiintoista. Niiden kehitystä on ollut mielenkiintoista tutkia, ja monet aiheet siirtyvät seuraavaan hyvin sulavasti. Lisäksi taustalla olevat odotukset ovat hyvä kannuste työn tekemiseen kunnolla.

Ehkä isoin asia, minkä opin opinnäytetyötä kirjoittaessa, oli materiaalin sulavuus ohjelmoinnin suhteen. Koodi vanhentuu hyvin äkkiä kirjamuodossa, ja jotkin komennot, jotka ennen toimivat, ovat voineet tulla korvatuiksi uusilla. Vaihdettuani verkossa olevaan tietokantaan, ongelmat katosivat hyvin pian, ja muuttuivat enemmän klassisiksi ja tutuiksi ohjelmointivirheiksi. Mutta kirjasta oli hyötyä muulla saralla: se tarjosi paljon paremman kuvan siitä, mitä kaikki yksittäiset koodin osat tekivät bottia varten. Materiaalin käyttö oli yleisesti paljon laajempaa kuin odotin.

Botit tulevat olemaan suuressa osassa kaikessa toiminnassa ajan kuluessa, kun koko ajan enemmän toimintaa automatisoidaan. Jo botin rakennusmahdollisuuksien määrä osoittaa, miten helppoa sellaisen tekeminen voi olla. Yksinkertaisista töistä eroon pääseminen tulee olemaan aina houkutteleva vaihtoehto, ja monet verkkoyhteisötkin arvostavat kehittäjää, joka osaa sellaisen botin rakentaa.

LÄHTEET

Chatbot Community. Medium.com. Viitattu 4.2.2020. 3 major problems with chatbots and chatbot development. https://medium.com/@chatbotcommuity/3-major-problems-with-chatbots-and-chatbot-development-503d84e176aa

Day, J. Tucker, B. 2019. Quora. Viitattu 17.6.2019. https://www.quora.com/Is-it-possible-for-a-single-database-to-be-used-by-multiple-websites-hosted-on-different-servers

Hintikka, K. Otavan Opisto. Viitattu 4.2.2020. Santra – Chattibotti. http://katehanke.fi/santrachattibotti

Keränen, M. 2018. Tekniikka & Talous. Viitattu 10.10.2019. Selvitys paljastaa: Tekoäly tuottanut vähän uutta liiketoimintaa Suomessa – ”yksittäistapauksia”. https://www.tekniikkatalous.fi/uutiset/selvitys-paljastaa-tekoaly-tuottanut-vahan-uutta-liiketoimintaa-suomessa-yksittaistapauksia/c2b55075-6a23-3f66-8260-f43253588160 .

Koivunen, S. 2019. Alma Talent. Viitattu 4.2.2020. Case-esimerkit valottivat bottien mahdollisuuksia Chatbot Day –seminaarissa. https://koulutus.almatalent.fi/blogi/asiakaspalvelu/case-esimerkit-valottivat-bottien-mahdollisuuksia/

Lobo, J. Fontseca J. Techcrunch. Viitattu 28.2.2020. Choosing the best language to build your chatbot. https://techcrunch.com/2017/12/20/choosing-the-best-language-to-build-your-ai-chatbot/

Raj, S. 2019. Building Chatbots with Python: Using Natural Language Processing and Machine Learning. Apress.

Rasa.com. Viitattu 3.7.2020 Tutorial: Building assistants. https://rasa.com/docs/rasa/user-guide/building-assistants/

Reilio, E. 2019. Kauppalehti. Viitattu 4.2.2020. Syrjäyttääkö chatbot ihmisen? https://blog.kauppalehti.fi/asiakaspalvelun-uusi-aika/syrjayttaako-chatbot-ihmisen

Salecha, M. Analytics India Magazine (AIM). Viitattu 4.2.2020. Story of ELIZA, the first chatbot developed in 1966. https://analyticsindiamag.com/story-eliza-first-chatbot-developed-1966/

Salecha, M. Analytics India Magazine (AIM). Viitattu 4.2.2020. Turing test: a key contribution to the field of artificial intelligence. https://analyticsindiamag.com/turing-test-key-contribution-field-artificial-intelligence/

Voskoglou, C. Towards Data science. Viitattu 28.2.2020. What is the best programming language for machine learning? https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7

Kohn, N. Medium.com. Viitattu 3.4.2020 Build an AI / Machine Learning ChatBot in Python with RASA — Part 1. https://medium.com/hackernoon/build-simple-chatbot-with-rasa-part-1-f4c6d5bb1aea

Kohn, N. Medium.com. Viitattu 3.4.2020 Build a simple ChatBot in Python with RASA—Part 2. https://medium.com/hackernoon/build-simple-chatbot-with-rasa-part-2-16726357b72c