University College London

MSc Computer Science

Faculty of Engineering

Department of Computer Science

A Conversational Chatbot Architecture for eHealth Systems

Author:

Niccoló Terreri

Supervisor:

Harry Strange

September 4, 2016

Disclaimer

This report is submitted as part requirement for the MSc Computer Science degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This project is about the architecture and design of the core backend of a conversational agent for eHealth applications. It is part of a larger team effort aiming at the delivery of a complete user-facing application, specifically modelled after Macmillan's eHNA system for cancer patients in the UK. It involved research in open source chatbot technologies, and related NLP tasks such as text categorization, as well as the design, testing and basic implementation of the system. The developement was carried out in weekly iterations in continuous consultation with a client for the University College London Hospital. The project delivered an extensible implementation in the form of a software package meant for use with an external system of delivery to the user (a webserver in the final product of the team effort).

Contents

Chapter 1

Introduction

> "Pointing will still be the way to express nouns as we command our machines; speech is surely the right way to express the verbs."

Frederick Brooks, 1995

1.1 The Problem

According to the 2014 National Cancer Patient Experience Survey National Report, only slightly over 20% of cancer patients across the UK reported having been offered an assassment and care plan specific to their personal circumstances over the past couple of years (Quality Health, 2014, p. 114). In an effort to increase the number of cancer patients who received such assessments, Macmillan Cancer Support piloted the Holistic Needs Assessment (HNA) questionnaire and health plan in 2008 (Macmillan, Holistic Needs Assessment). This is a self-assessment questionnaire where the patient identifies what their concerns are from a range of

personal, physical, emotional and practical issues they may be facing in their lives in relation to their condition. The completion of the questionnaire is followed by the creation of a care plan through a consultation with a clinician, with further advice and referrals as needed. Macmillan began trialing an electronic version of the questionnaire in 2010, progressively extending provision of the eHNA to more and more sites (Rowe, 2014).

This project is a about the use of an intelligent conversational system to gather further information about the patient's concerns through an electronic self-assessment tool, ahead of the creation of a patient care plan. This is primarily an attempt at introducing the conversational User Interface in electronic health applications generally, investigate related natural language processing and tasks, and in particular explore the applicability of computer advisors to Macmillan's eHNA in a growing effort to improve the quality of support cancer patients receive across the UK.

Intelligent conversation systems have enjoyed an increasing amount of media attention over the last year (numerous articles among which: The Economist, 2016; Berger, 2016; Knowledge@Wharton, 2016; Finextra Research, 2016; Yuan, 2016; French, 2016). With applications of artificial intelligence to using natural language inputs for different purposes, including general purpose mobile device interfaces (Viv, 2016; Dillet, 2016). Furthermore, several technology companies have started offering "Artificial Intelligence as a Service" products. Among these are BloomsburyAI (founded at UCL) and bespoken companies such as Google and Microsoft (Pandorabots, 2016; Riedel et al, 2016; Microsoft Cognitive Services, 2016). This appears indicative of the fact that chatbot and natural language processing technologies have reached a level of maturity comparable to that achieved years ago by haptic technology, that we find almost ubiquitously in human-computer interfaces and everyday use of computing devices today.

This project is part of PEACH: Platform for Enhanced Analytics and Computational Healthcare (PEACH, 2016). PEACH is a data science project that originated at University College London (UCL) in 2016 that sees Master level candidates working together on the data platform and on related projects. With more than twenty students across multiple Master courses, it is one of the largest student projects undertaken in recent years at UCL, and it is part of a long-term strategy to bring the UCL Computer Science department and the UCL Hospital closer together.

The scope of the present report is limited to the architecture and implementation of the chatbot system, as opposed to a complete user-facing product: the complete application is a joint effort of four members of PEACH, with distinct concerns being assigned to different members of the team. The author of the present document is tasked with design and implementation of the core system backend. The other members of the chatbot team include: Andre Allorerung (MSc SSE) as the technical team lead who also oversees of the integration of the system with the resources available to PEACH and the data storage system that will persist information gathered through the chatbot system. Rim Ahsaini (MSc CS) working on a specialized search engine for resources that may interest and help support cancer patients based on their concerns (to be available both through conversation with the chatbot and independently), Deborah Wacks (MSc CS) as lead UX designer working on the implementation of a webserver through which deliver the chatbot and search engine to users.

1.2 Project goals and personal aims

The main project goal is the delivery of a basic but easy to extend and modify chatbot software system, specifically targeted at assisting with the identification

and gathering of information around cancer patient issues, modelled after the Concerns Checklist (CC) electronic questionnaire form (NCSI, 2012). Finally, one of the major challenges with eHealth problems is represented by having to hand confidential patient data (as will be discussed in Chapter 2 of this report). Summarily:

- Design and implement a chatbot architecture tailored to the issues surrounding software systems in healthcare (in particular around treatment of sensitive patient data)
- To integrate with a specialized search engine (developed by another member of the team)
- To explore other applications of NLP that could be useful to extract information from natural language data.
- To implement a chatbot brain using open source technology.
- To develop the system with Macmillan eHNA as the main reference.

Personal goals of the author include:

- Learning Python in an effort to gain exposure to a new programming language
- Leverage the author's background in computational linguistics, and explore the field of natural language processing
- Learn about applications of machine learning to natural language processing
- Improve software engineering skills by applying best agile methodology practices

1.3 The project approach methodology

An agile methodology approach was adopted for the project, in line with the author's stated interests. This meant maximizing time spent outside of meetings, save for where communication between team members and others was required. The project was paced in weekly iterations where aspects of the system to implement would be selected from a backlog to be delivered for the next week, in consultation with Dr Ramachandran who acted as the client for ever project connected with PEACH (Beck and Andres, 2014, pp.46-47). Great emphasis was also put on testing as part of deveopment, in particular the discipline of Test Driven Development.

A top-down system design and implementation was also adopted, with the next largest system abstraction being priorized first in order to always have a working system being progressively refined. These methodology guidelines where established in accordance with the reccomendations of Brooks (1995, pp.143-144, 200-201, 267-271), Martin (2009, pp.121-133; 2003, chapter 2, 4, 5) and Beck (Beck and Andres, 2004; Beck et al, 2001).

1.4 Report overview

This report is structured as follows:

- Chapter 2 provides more extensive background into the NLP and chatbot open source resources that were explored.
- Chapter 3 describes the requirements as gathered through the contacts in healthcare and the Macmillan charity available to PEACH.
- Chapter 4 details the system architecture, design and the implementation,

highlighting its current limitations and design.

- Chapter 5 discusses the benefits of TDD to systems design, how system testing was done as part of development, and the evaluation of the machine learning component of the system.
- Chapter 6 concludes with an evaluation of the project results, a review of the effectivenss of the core tools used, and reccomendations for the direction of future work on the system.

Chapter 2

Background Research

This chapter details the literature review for the project: the background reading, and the tools and frameworks selection process.

2.1 The electronic Health Needs Assessment questionnaire

The primariy already existing reference for the software to be built by the team is Macmillan's eHNA.

2.1.1 Macmillan Cancer Support

Macmillan Cancer Support developed the eHNA for the purpose of extending the range of cancer patients in the UK covered by individual care plans, made with the individual's very personal and unique concerns they incurred into in relation

to their condition. These concerns are gathered through variants of an electronic questionnaire offered by Macmillan to selected trial sites. Paper versions and variants of the questionnaire existed before the introduction of the eHNA in 2010, and have been in use since before then (Watson, 2014; Brittle, 2014; NCAT, 2011).

The electronic questionnaire is designed to be carried out on site mostly through haptic devices (such as tablets), just ahead of meeting the clinician that will help draft a care plan for the patient. There is the option to complete the questionnaire remotely, although the adoption of this alternative is made difficult by the work habits of key personnel, who are used to providing a device to the patient in person and ask them to carry out the questionnaire while at the clinic.

The patient uses device touch interface to navigate through various pages selecting concern categories from a predefined list. There are several versions of questionnaires available, modelled after the various paper versions, depending on which one the clinic previously used.

Patients typically select three-four concerns (up to around six, mostly depending on the type of cancer they have). The questionnaire takes on average less than 10 minutes to complete. The front end of the system is implemented as web-app. Access to the assessment is restricted to scheduled appointments that clinics set up for individual patients, either via delivering the questionnaire on the clinic site, or, if the questionnaire is carried out remotely, via use of a one-time 6 digit PIN number, alongside the patient's name and date of birth.

2.1.2 The Concerns Checklist

Given the variety of different versions of the questionnaire, the team was advised to focus on the one that is most commonly used: the Concerns Checklist (NCSI, 2012).

In this version of the questionnaire, the patient selects their concerns from a range of more than 50 individual issues, each falling into one of 10 categories, and selects a score for it in a range from 1 to 10. Each category may itself be a subcategory of the following major topics (see Appendix A for a full list):

- Physical concerns
- Practical concerns
- Family concerns
- Emotional concerns
- Spiritual concerns

2.2 Patient Data for Research in the UK

As mentioned in the project goals section in Chapter 1, handling confidential patient data poses particular challenges to eHealth related projects. Just before the start of the project, the whole PEACH team underwent training about handling patient data and the relevant legislation in the UK. These affected design and implementation decisions.

Generally speaking, authorization is required before any information provided by the patient can be used in any way except the specific purpose of their healthcare (Data Protection Act 1998). It may be possible to make use of third party services

provided the data has been fully anonimized and cannot be linked back to the patient. Preferrably, a special agreement (such as a Data Transfer Agreement) should be brokered to ensure both parties understand the legal and ethical implications of sharing even anonimized data (in such cases, the duty of confidence, UK common law, does not extend over to the third party). Note however, that it is sometimes difficult to ensure that data has been anonimized, even by removing all information considered personal under UK law: for example, if a person happens to have a rare disease, or information about the geographic location of the patient can be retrieved from the data being shared with the third party.

For this reason, the implementation of the current project does not share any of the data extracted from user input with external third parties although the architecture will allow for such a choice to be made in a future iteration of the chatbot project.

2.3 The Chatbot

The very high level general architecture of a chatbot system is normally described in terms of (Bird et al, 2014, Chapter 1 section 5.5; Jurafsky and Martin, 2009, pp.857-867):

1. A natural language understanding layer
2. A dialogue and or task manager
3. A natural language generation layer

The first may be simply implemented through hand-written finite state or context free grammars. The system takes in the user input and attempts to parse it according to predefined grammar rules in order to extract the relevant information (Jurafsky and Martin, ibid, p.859).

The simplest way to address number 3 is to have a set of hand-written natural language templates that are filled in with relevant information before being output to the user. A way to avoid having to hand-write the grammar rules is by making use of a probabilistic parser that also seeks to fill-in slots of information required by the system to carry out the task (ibid. p.860). Finally, the dialogue or task manager would be what models the information required of the user for the completion of the task, and manages the turn taking and the grammar rules to match for the user inputs and the templates to use in output.

This type of rudimentary system can be contrasted with more complex systems that include modelling of the conversational context and understanding and generation modules that can operate with higher level abtraction than mere patterns of symbols (like the grammar rules and templates described above), which we may want to call "dialogue acts". These more advanced systems are sometimes referred to "information-state" systems as opposed to "frame-based" systems that coerce the conversation to a very specific task (Jurafsky and Martin, ibid, pp.874-875).

For the purposes of the problem at hand, it is difficult to decide which of these would be ideal. For one, the tight control over the topics in the conversation provides a good way to ensure the user does not get distracted into trying to ask the system questions that are irrelevant to the gathering and extraction of user concerns. On the other hand, especially where the system is being used without previously gathered knowledge about what the categories of concerns of the user are, it may be more suitable to have a general purpose "information-state" system, capable of carrying out a conversation with the user.

2.3.1 The Chatbot "Brain" Market

This section aims at describing the options that were discovered while researching ways to set up an initial chatbot. Open source projects available mostly make use of the patter/grammar and template model described above, but there is a notable alternative in emerging (yet immature) technology using neural networks or similar tools to build conversational models. Finally, there are AIaaS providers: providers of remote intelligent agents and related services (Pandorabots, 2016; Riedel et al, 2016; Microsoft Cognitive Services, 2016), similarly to the web hosting solutions of Amazon or Microsoft (AWS, 2016; Azure 2016). For the reasons outlined in the above section on the particular legal issues around the problem domain, it was deemed unfeasible to use external services that would host the chatbot service and as a consequence receive and process patient information (even in anonimized form).

Exploring the open source chatbot "market" it is easy to appreciate how this world has mostly been evolving outside of academia. The main sources for this section of the report are the individual websites of the tools explored, and the forum of https://www.chatbots.org/ai_zone/ and related readings (Morton, 2011; Wilcox, 2011).

In the architecture model proposed above, these systems make it easy to write documents that define the patterns and templates of a "frame-based" system, simplifying the process to build all three of the above defined layers. We now proceed to review the various options, and motivate the choice of tool used in this project.

The Artificial intelligence Markup Language

AiML is a version of the Extensible Markup Language (XML) that was specifically designed around providing a framework to define rules, patterns and grammars to match user inputs to appropriate templates. The language was created with the objective of providing a transferrable standard. On top of the basic patterns and template, AiML provides ways to use wildcards or optional sub-patterns in the input pattern and to capture parts of the user input for processing or to repeat back to the user by decorating the template.

AiML also provides ways to define topics as restrictions over the set of matchable patterns. Entering a topic effectively means restricting the patterns that user input can match to the ones associated with the topic. It is also possible to set and read internal variables tied to one user, and use this state in conditionals to decide which template to use in the output; it is possible to refer back to the previously matched input, for example to read a follow up to a yes-no question (see http://www.alicebot.org/aiml.html; Wallace, 2014).

AiML is only a standard for defining this information, and there are separate guidelines to follow to implement a AiML reader (or "interpreter"). The set of files making up the AiML "bot" are commonly called the chatbot "brain" and there are a number of interpreters available for AiML in various programming languages. There are freely available "libraries" of AiML files for others to include into their own chatbot[1] (Wallace, 2011).

1. https://github.com/pandorabots/rosie/,
http://www.alicebot.org/downloads/sets.html,
http://www.square-bear.co.uk/aiml/.

RiveScript

RiveScript is an alternative standard to AiML the objectives of which are to be as expressive and useful as AiML, but with a simpler syntax, getting rid of the XML (Petherbridge, 2009; Petherbridge, 2012; https://www.rivescript.com/compare/aiml). Like AiML, RiveScript has support for topics, remote procedure calls, conditionals, redirections and other features. One thing that will be particularly relevant for the remainder of the discussion is the possibility to define "arrays" or sets of equivalent terms. This seemed useful to define synonyms and allow certain patterns of user expression to be captured if they matched some synonym.

But what made RiveScript particularly attractive was not just its simpler syntax, it was rather the fact that there is an official Python RiveScript interpreter but (at the time of writing) there are no easily traceable AiML interpreters implemented in Python (Petherbridge, 2016). The best candidates are pyAIML and pyAiml-2.0 (Tomer, 2014; Iaia, 2016), neither of which has either been maintained for a long time, or is very stable. This also applies to most other open source AiML interpreters implementations at the time of writing, leaving only a couple standing (ALICE A.I. Foundation, 2014; Morton and Perreau, 2014).

ChatScript

ChatScript is in many ways similar to RiveScript in that it instead of extending XML it wishes to have a very easy to read syntax (Wilcox, 2011; Wilcox, 2016b). In ChatScript, it is possible to define "concepts" like RiveScript "arrays". ChatScript also is also integrated in WordNet: a lexical database for the English language that primarily models synonimity and hyponimity between English words (Fellbaum, 2005).

ChatScript, like the others, supports external procedure calls, wildcards, optional sub-patterns and the other pattern matching features of the previous standards. Something that distinguishes ChatScript from the other standards examined is that there is only one interpreter implementation in C++ and no other open source interpreter projects.

SuperScript

SuperScript is a fork of RiveScript with syntax elements inspired by ChatScript (Ellis, 2016; Ellis 2014). It boasts features from all of its precedessors, including WordNet integration, plus a complex input processing pipeline that will attempt to analyze the user input as a question and try to provide an answer to it, in light of the preceding conversation.

The core issue with this system is the fact that it is only made for NodeJS, in particular, only versions 0.12 or 0.5x. While the author is personally unfamiliar with Node, this came across as a red flag. The recommended version of NodeJS for most users at the time of writing is 4.5.0, while the latest build version is 6.4.0[2].

This may create problems where this project is used in conjunction with NodeJS in other applications (on the webserver for example), and while there are workarounds to having to keep multiple versions of Node, there is the risk of making it more and more difficult to maintain the system as Node and SuperScript evolve (see Mota, 2016 here for how to manage multiple NodeJS versions: https://www.sitepoint.com/quick-tip-multiple-versions-node-nvm/). Secondly, given the stated personal

2. Node Core Technical Committee and Collaborators, 2016;
https://github.com/nodejs/node/blob/master/doc/changelogs/CHANGELOG_V4.md#2016-08-15-version-450-argon-lts-thealphanerd,
https://github.com/nodejs/node/blob/master/doc/changelogs/CHANGELOG_V6.md#2016-08-15-version-640-current-cjihrig

aim of the author to explore the Python programming language, the choice of a system only meant to work with JavaScript made it a less than ideal candidate.

Neural-network-based conversation models

Work has been done to use various types of neural networks to produce general purpose conversational systems. Sordoni et al (2015), for example used several recurrent neural networks to keep account of the conversational context by modelling the information previously processed (ibid, section 3). Shang et al (2015) use Statistical Machine Translation techniques, treating the response generation (or "decoding") phase of the process as a linguistic translation problem (ibid, figure 1).

Vinyals and Le (2015) have used the seq2seq (sequence-to-sequence) model that uses a recurrent neural network to map an input sequence to an output sequence token by token with interesting results. However, this is, as they claim, a purely data-driven approach that relies on a significant volume of pre-existing data to train the model. No such data exists for the specific domain of the present project, and therefore would probably only be possible when sufficient natural language conversation data specific to the system domain has been gathered (or alternatively generated).

Conclusion

AiML and competitors all seem to sport the same array of basic features, but of particular interest for the current project was the possibility to define and control the content of topics, in order to provide only domain-relevant replies from the system to the user. Of the four, RiveScript is the only one that explicitly supports

topic inheritance, which seemed useful with respect to creating a hierarchy of macro and micro topics: for example, having a global scope with general purpose commands (such as change topic) with subscopes like "family" and "physical" which could be further subscoped to have issue-specific matchers, such as matchers that are only relevant to respiratory problems and would not occur in the related physical category of nausea problems, although both would share some general matchers about physical issues (Petherbridge, 2009, here: https://www.rivescript.com/wd/RiveScript#topic). ChatScript also allows "enqueuing" of topics with the concept of "pending topics" and also control of context via "rejoinders" (Wilcox, 2016a, pp.9–; Wilcox, 2016b, pp.5–).

Another point of interest (again, given the author's aim to explore Python) is the open source software available for use with the project. Given the considerations already provided, SuperScript seemed like the least comfortable option from this perspective, with ChatScript (C++) being second least. This would leave RiveScript and AiML, with RiveScript's simpler but expressive syntax being the final deciding factor for the current implementation.

2.4 Natural Language Processing

As stated in Chapter 1, part of the author personal aims included to learn about NLP and leverage the author's background in computational linguistics. Specifically, the possibility to classify user input according to the topic being mentioned was identified as a useful application of NLP to the problem. It may be useful to add tags to user inputs to be matched within a chatbot brain that would otherwise be unable to generalize. This would effectively represent a hybrid model where both a machine learning component and a "rule-based" (input-patterns) approach are

used as part of a more complex system[3].

2.4.1 Text Classification

Text classification is the NLP task of assigning a category to an input from a predefined set of classes (Sebastiani, 2002, p.1; Manning et al, 2009, pp.256-258; Manning and Schütze, 1999, pp.575-576). Particular to our case, the documents will be natural language conversational user input, and the set of categories will be the macro cateogries of issues that have been extracted from the concerns checklist (CC) version of the questionnaire (see above). This task is turned into a supervised machine learning task, by training a model over a set of document-category pairs (Sebastiani, 2011, slide 7, 13).

The internal representation of each document to the classifier is a sparse vector representing the features or characteristics of the document relevant to the classification task. These features can be, for example, the occurrence or non-occurrence in the document of certain words (set-of-words approach) (Sebastiani, 2011, slide 66; Manning et al, 2009, pp.271-272)[4].

2.4.2 Open Source NLP Libraries

The open source NLP libraries that were considered as part of the project were the ones that could easily be used with the Python programming language (in line

3. Generalization is characteristic of approaches to artificial intelligence resembling experience as a way humans acquire knowledge (Russell and Norvig, 1995, pp.163-165, 592; Biermann, 1986, pp.134-135; Hawthorne, 2014; Beall and Restall, 2014; Hume, 1777, section 4, part 2).

4. The task of sentiment analysis was also deemed important to the assessment of the user distress level with respect to issues the user would be discussing with the chatbot. But became less important as requirements where clarified (Chapter 3). The reader is directed to Liu, 2010, p.628, 637–; and Ding, Liu and Yu, 2008 for more information on the topic.

with the author's personal aims), so long as the open source tools available for Python proved sufficient for the project purposes. This excluded, for example, the OpenNLP Apache library, due to its focus on Java (Apache, 2015). The second desideratum was for all PEACH subprojects involving some degree of NLP to use the same family of technologies and open source packages. This was meant to make it easier to reuse results from the current iteration in the future and build a common base so that the different subprojects may draw from each other work.

It was decided, primarily based on the experience of the memebers of the PEACH team that had previously worked with NLP to use the Natural Lanugage Tool-Kit (NLTK) as a baseline, but to not be afraid to adopt other tools as needed by individual projects (Bird et al, 2014). The decision was also made on the basis of the expected needs of the individual subprojects. Its focus on Python also made it a better solution with respect to other suites such as the Stanford CoreNLP, due to lack of extensive Python bindings from the Java implementation (Manning et al, 2014; Smith, 2014). The present project thus made primarily use of the NLTK, and packages built on top of it.

The NLTK exposes a range of natural language corpora and ready available implementations of various types of classifiers, exposing an intuitive API (Bird et al, 2014, Chapter 6 see also Manning et al, 2009, pp.271–). An obvious problem with the present project is that no data of the relevant shape was available whatsoever. See Chapter 5 and Appendix C for more information on the solution adopted.

2.5 Generating Chatbot Brain Data

Given the conclusion of using a software package that works based on input patterns matchers and output templates, which have to hard coded, investigation began into automated generation of matchers and templates. One area that it became clear early on could benefit from automated generation was with patterns to match not against a particular set of terms, defined inline into the pattern, but English words close in meaning.

2.5.1 The word2vec Algorithm

One way to automatically generate synonyms is by looking at regularities discovered in the use of English words through unsupervised learning. This is at the core of what the word2vec algorithm does: it discovers these regularities based on the position words are used in sentences. For each word in the training data (the vocabulary) the algorithm constructs a vector representing the positional regularities discovered in the training data.

Similarities between the use of words can be then expressed in geometric-algebraic terms as the cosin distance between vectors representing the words (Mikolov, 2015; McCormick, 2016b). This sort of similarity seemed like an interesting way to automatically generate synonyms for use with the chatbot.

2.5.2 The Gensim Library

The Gensim library (Rehurek, 2014; Rehurek and Sojka, 2010; see here: https://github.com/RaRe-Technologies/gensim) is another natural language processing

tool available for use with Python specialized in document similarity computations and related tasks. It seemed straightforward to use word2vec for the state purpose in combination with Gensim (McCormick, 2016a).

Since the sort of data relevant to the training of a word2vec model for the purpose of synonym generation did not require domain-specific data, but was in fact best gathered through general English sources, the model that was used for the synonym generation task was a model that had been pre-trained over a significant amount of Google News data (McCormick, 2016a). See Appendix C for results and evalutation.

2.5.3 Alternatives

WordNet (Fellbaum, 2005) is perhaps a better alternative. There are ways to interface with it via the NTLK (see http://www.nltk.org/howto/wordnet.html), but, as emphasized in the chatbot brain discussion above, some chatbot frameworks other than the one used for the current implementation (RiveScript) already offer similar WordNet integrations out of the box (see for ChatScript: Wilcox, 2016b, pp.10-11). For other options into using dictionary definitions to extract synonyms the reader is redirected to Wang and Hirst, 2012.

2.6 Conclusion

This concludes the preliminary investigation section of the report.

Chapter 3

Requirements Gathering

This chapter describes the full problem statement, and the way the list of requirements was produced and agreed on by the stakeholders. Recall that the scope of this project is the architecture of the core system, not a full end-to-end implementation, which is the objective of the PEACH chatbot team. UI/UX design does not fall within the scope, and neither does a data persistence strategy (datastore solutions etc) nor a full deployment plan.

3.1 Building the Right System

The problem to be solved is an introduction of the conversational UI into the eHealth sphere taking into account UK legislation over the use of patient data. This is inspired by the Macmillan Cancer Support chartiy eHNA system, that is just finishing its extended trial period and aims at extending the coverage of support cancer patients receive in the UK with personalized care plans (Mac Voice, 2014). The introduction of this UI is intended to help streamlining the process of

filling out the eHNA for the patients, and gather insight into the patient issues ahead of the in-person review, shortening the time needed for the creation of the care plan in oder to facilitate its creation, extending the coverage to more cancer patients, in an effort to improve the quality of care they receive across the UK.

In this initial phase of the project, the key objective is the engineering of an extensible architecture, which is expected to be subject of significant modifications as the software is tweaked in following iterations to accommodate for changes in the nature of the problem, and in the technological landscape. In other words, focus on minimization of technical debt (Fowler, 2003).

As discussed in the previous chapter, there are increasingly interesting developments happening in the use of recurrent neural networks for chatbot applications, but at the time of writing the technology is still immature, while an older model primarily based around the gathering of specific set of parameters from the user through conversation, achieved through hard-coded input patterns matchers and response templates, that has matured over the last fifteen years, is already being used in professionally developed software (Vinyals and Le, 2015; Jurafsky and Martin, 2009, Chapter 24; Wilcox, 2011).

The aim of the project is therefore to engineer this architecture and provide a basic proof-of-concept implementation of the system, with *the core of the chatbot system being an easy to replace implementation detail* rather than the core focus of the project effort.

3.2 Requirements Gathering

The project was first proposed by Dr Ramachandran, who also organized meetings between the chatbot team and Macmillan staff (including the technical lead of the Macmillan eHNA, Andrew Brittle). The original plan was to have both the author and Rim Ahsaini working on the core chatbot system, with the full questionnaire being completed by the user through interaction with the chatbot.

During the first meeting with Macmillan, the idea of a specialized search engine was presented, with Rim Ahsaini interested in making the implementation of that system her project. From there came also the idea of integration between the two systems, with the chatbot system having to decide whether to make a query during the interaction with the user.

During the second meeting, when the team was given a full demo of the eHNA system. It emerged that users tend to only select a relatively small amount of concerns, between two and six depending on what type of cancer they have. Furthermore, the questionnaire itself takes a relatively small amount of time. The requirements for the core chatbot system were afterwards revised to allow the user to first fill out the questionnaire before talking to the chatbot in order to keep the efficiency of the eHNA software model. This made certain tasks, such as sentiment analysis, less useful for the project, given that the user would independently specify their level of distress.

The rest of the requirements were mostly decided by the author and project supervisor on the basis of what would be most useful to investigate from a technical point of view looking at the future of the project. Objectives were decided on a weekly basis, in line with agile methodology. Table 3.1 lists the requirements gathered according to the MoSCoW method.

3.3 Use Cases

Given the role of the project in the larger team effort, use case modelling was considered inadequate to describing the project requirements, and more suited to a user-facing system. Therefore, no use cases were produced.

Table 3.1: Requirements table

Label	**Requirement**	**Priority**
RQ0	The chatbot will categorize userinput into predefined concern categories	Must
RQ1	The chatbot will generate replies aimed at asking the user for more details, and gather information on concern raised	Must
RQ2	The chatbot will decide whether the generative reply should be provided as the result of a query or bot-brain generated reply	Should
RQ3	The chatbot will model the data gathered appropriately for durable storage in datastore	Must
RQ4	The chatbot will interface with the search/query engine component of the larger system it is part of	Should
RQ5	The chatbot may performsentiment analysison user input	Could
RQ6	The chatbot may use the preprocessed input (categorization/sentiment analysis) to inform the bot-brain	Should
RQ7	The chatbot may interface with server side request handling logic	Should
RQ8	If a rule-based approach is taken, steps should be taken to automate the generation of rules	Should
RQ9	The chatbot mayreceive/return queries in the form of JSON/XML documents	Could
RQ10	The chatbot may be deployed outside the context of eHNAs as an independently accessible service	Could
RQNF0	The chatbot will use open source technologies instead of remote APIs due to the sensitive nature of patient data	Must
RQNF1	To use Macmillan Cancer Support's eHNA and the Concerns Checklist questionnaire as references for the project	Should
RQN0	The chatbot will not have an offline/cached operating mode	Will not

Chapter 4

System Design and Implementation

> *"Out of all the documentation that software projects normally generate, was there anything that could truly be considered an eningeering document?" ... "Yes, there was such a document, and only one–the source code."* **J. Reeves, 2001**

This chapter describes the most interesting aspects of the delivered system architecture and implementation. First, the design principles followed during development are described, then a high level overview of the project structure is provided before moving on to more detailed descriptions of individual subpackages.

As mentioned before, the complete application is the product of a team effort, a web server and specialized search engine being separately developed, while the author focuses on the core chatbot brain. The overall architecture of the components is illustrated below (Figure 4.1).

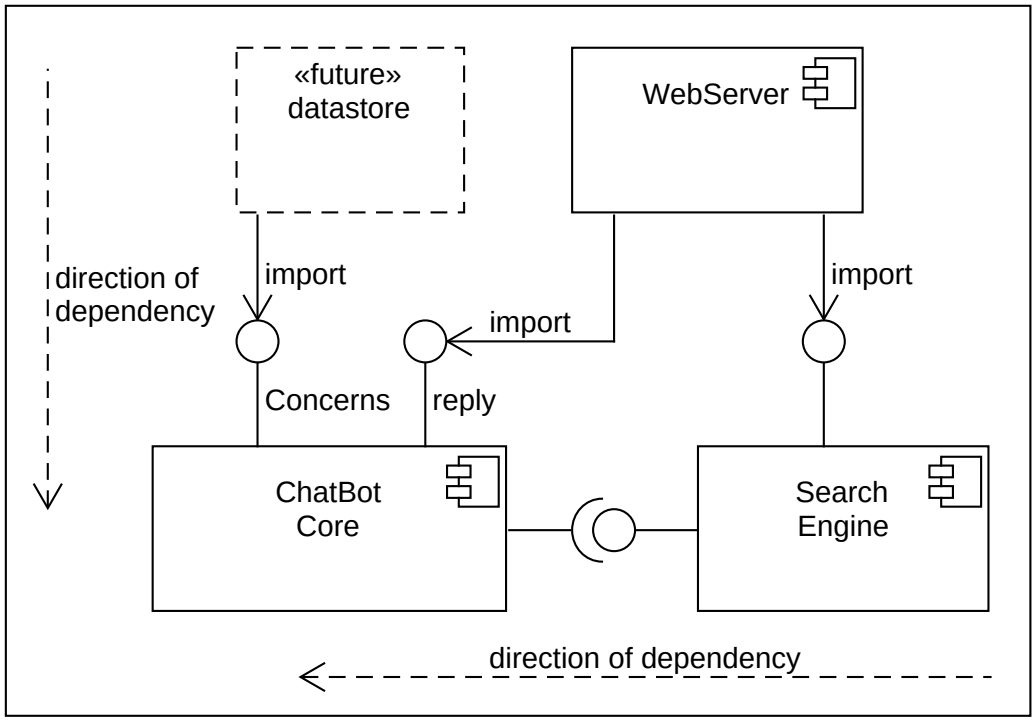

Figure 4.1: System component diagram

4.1 Software Architecture

4.1.1 Principles of Software Design

SOLID principles of software design were followed in the design of the system; these are dependency management principles (policing the "import" statements throughout the project) and clean design principles (Martin, 2003, Section 2). Additionally, clean code design guidelines (Martin, 2009; McConnell, 2004, chapter 5) were also followed. The chief purpose of these efforts is to design the system for change, and make the code readable to *humans* (Martin, ibid, pp.13-14). This approach leads to extremely specialized and short class bodies and functions, and the separation of different levels of abstraction. Furthermore, the project was structured with the "plugin model" architecture: ensuring that the direction of dependency flow in the direction of the core of the system (Figure 4.1).

The core of a chatbot system are the high level abstractions precisely defined in the interfaces and the abstract classes (Dependency Inversion Principle: Martin, 2003)

4.1.2 High Level Structure

The programmer accessing the package structure should be able to understand what the purpose of the system and of the modules within is without further assistance (Martin, 2011). The architecture diagram details the high level blocks of abstraction in the system (Figure 4.2). We will look at the core message processing logic of the system first, then discuss the RiveScript brain implementation, then move on to the data models and conversation management, and finally the machine learning components of the system.

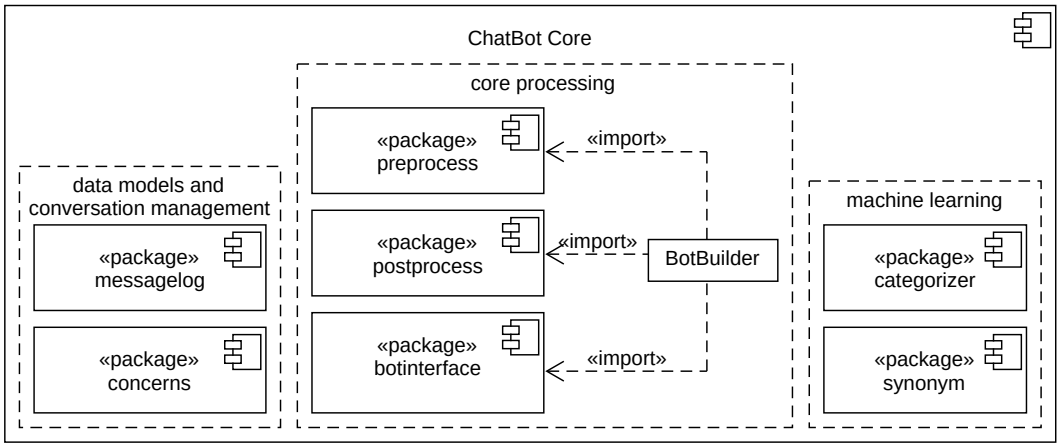

Figure 4.2: Architecture diagram

4.2 Core Processing

User messages are piped through a preprocessing layer that filters them to simpler forms in order to simplify the pattern matching in the chatbot brain. The reply from the brain is also postprocessed in order to decorate reply templates with results from the search engine when needed. We will look at each layer in detail.

4.2.1 The BotInterface

Within the botinterface subpackage we can find the core high level abstractions of the system: the bot_abstract module defines the interface to the bot, with a very restrictive set of methods that essentially make the abstraction a simple "response machine". Given a message, an appropriate natural language reply is expected[1].

1. While dynamically typed languages (such as Python) do not require inheritance for polymorphism, having the interface clearly defined help specify the expectation to other programmers. Therefore, interfaces are specified and inherited from in the code.

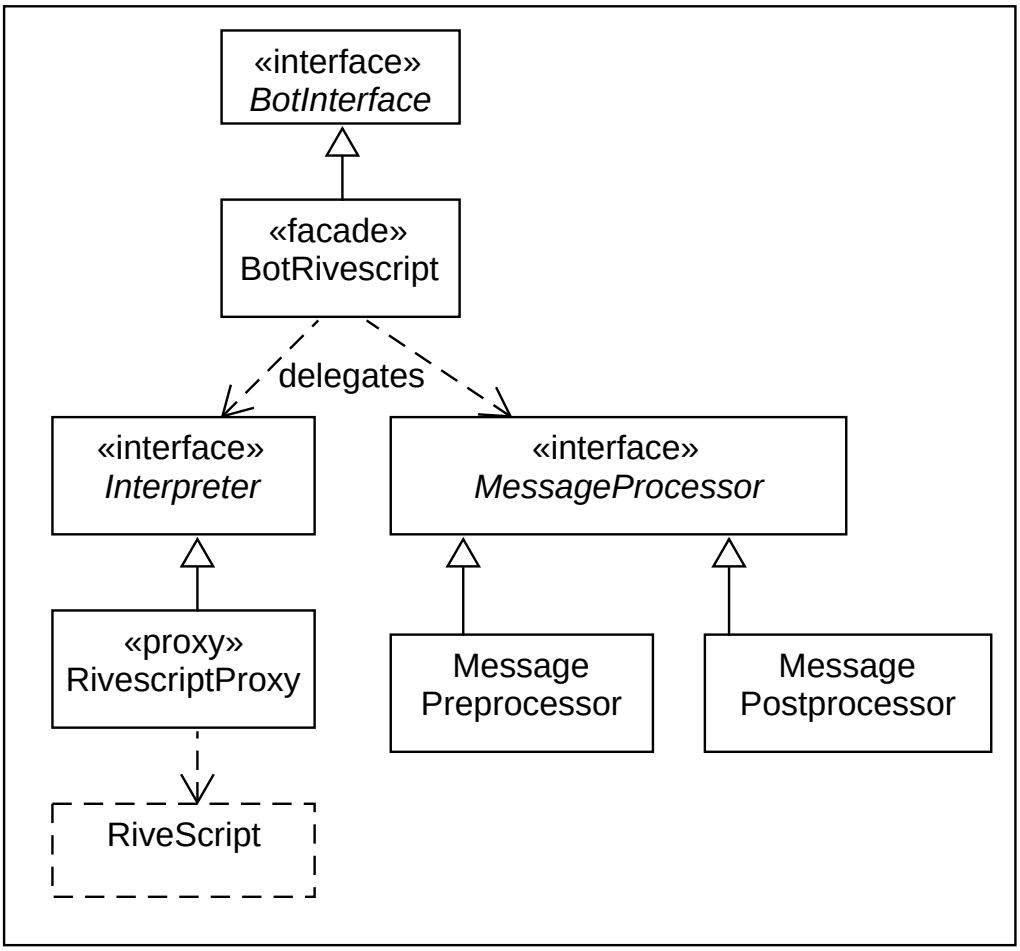

Figure 4.3: Core processing class diagram

Looking at the the concrete subclass of this interface, BotRivescript, we find it is in fact a FAÇADE: it essentially delegates the processing of the natural language message to other components of the system. Specifically, it delegates to a message preprocessor to process the input to the system and a postprocessor to process the system output through a MessageProcessor interface both of these conform to. The modularity of the design makes it easy to change implementation of these delegates, and provides a clear separation of concerns with the message coming into the system being preprocessed prior to being forwarded to the message interpreter through a PROXY, and then postprocessed as needed (Gamma et al, 1995, pp.185-193, 207-217; Martin, 2003, pp.327–).

This provides a degree of decoupling from the chatbot package, instead of making it a central component of the system, allowing it to be changed for another one of the options surveyed in Chapter 2 with relative ease (see Appendix A for more details). Furthermore, the succession of assignments within the public "reply" method serve to clarify the process to the (human) reader and also enforce proper temporal succession by requiring the next method in the sequence to take in as argument the return value of the previous method (Martin, 2009, pp.302-303). This pattern is repeated elsewhere throughout the project:

```
#from botinterface/bot_rivescript.py
def reply(self, message):
      userid = message.getUserid()
      messagecontent = self._preprocess(message.getContent())
      reply = self._interpreter.reply(userid, messagecontent)
      reply = self._postprocess(reply)

      return reply
```

BotBuilder

The bot_builder module is responsible for creating the concrete instances of message pre and post processors and their dependencies. Creational duties are slightly "special" in the sense that they require explicit dependencies on concrete classes and modules that define the constructors for the objects that will be used throughout the system (Dependency Inversion Principle, see Martin, 2003). To clarify the meaning of "delegates" in Figure 4.2 and others in this chapter: the BotBuilder will wire up all of the relevant dependencies, so that the façades do not have to instantiate the concrete subtypes of the interfaces themselves.

4.2.2 The Preprocessing Layer

The preprocessor is in turn another façade, like the BotRivescript class: it delegates to stopword removers, tokenizers and stemmers to normalize and simplify the user input so that it may be easier to match against the patterns specified in the chatbot framework grammar (Figure 4.4). This helps with RiveScript, as certain input patterns that should be matched do not match if the user mispells a word, uses a declination of a verb or adds stopwords (such as determiners) which had not been anticipated in the grammar.

Modules responsible for the preprocessing are defined within the "preprocess" subpackage. Notice how the dependency to the external nltk package is limited to only the concrete implementations that makes use of it in this class hierarchy. Just like with proxying the chatbot package, the intention is to not tightly couple the core of the system to the external library, but to keep the coupling loose, so that in the future other tools or solutions may be used instead.

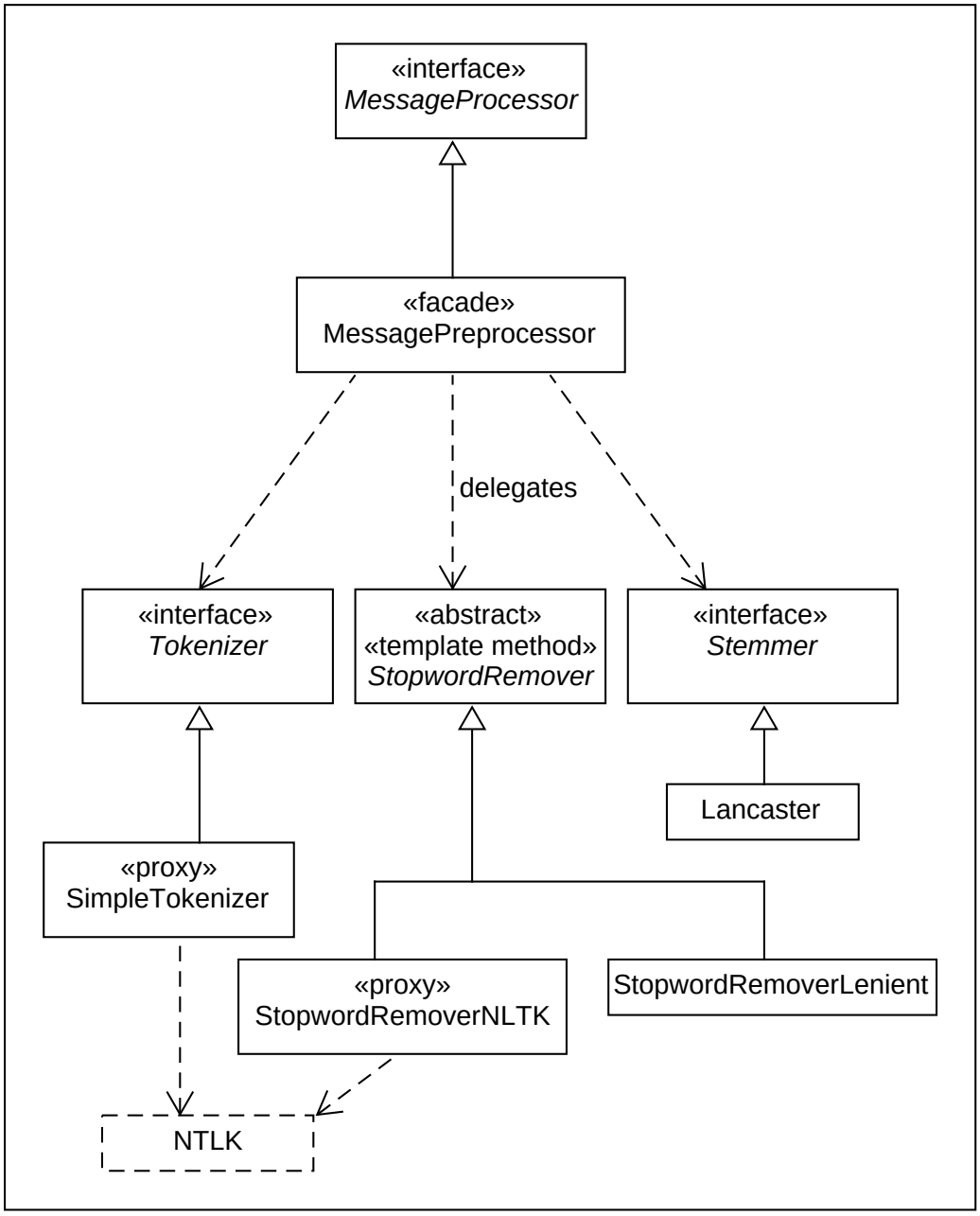

Figure 4.4: Preprocessor class diagram

More interesting is the application of the TEMPLATE METHOD pattern in the StopwordRemover abstract class (Gamma et al, 1995, pp.325-330). The stopword removal algorithm expressed in the StopwordRemover abstract class relies on subclasses exposing an iterable defining the stopwords to remove from the tokens. The set of stopwords defined in the nltk is too strict and would remove semantically meaningful tokens such as "no", "not" and declinations of "to be" (presumably on account of the fact that this is also used as an auxilliary verb and would create noise in information retrieval tasks). Therefore, a more lenient implementation was provided and used in the system.

4.2.3 The Postprocessing Layer

The postprocessor, inheriting from the same MessageProcessor interface as the preprocessor, is also a façade. The role of the postprocessor in the current implementation relates to the integration with the external search engine component (Figure 4.5).

The modules it defers to extract a keyword from the system output message, perform a query with that keyword and then decorate the message with the result before returning this to the main façade. We can see that the postprocessor subsystem is also defined within its own subpackage, currently using simple implementations aiming at extracting a single keyword from the system output message, then putting a single search query result back into the message.

This system was created ahead of the completion of the external search system, and was therefore built based on the author's assumptions about the information required and returned by this other system. The employment of an ADAPTER pattern sitting at the boundary between the systems enforces compliance with the

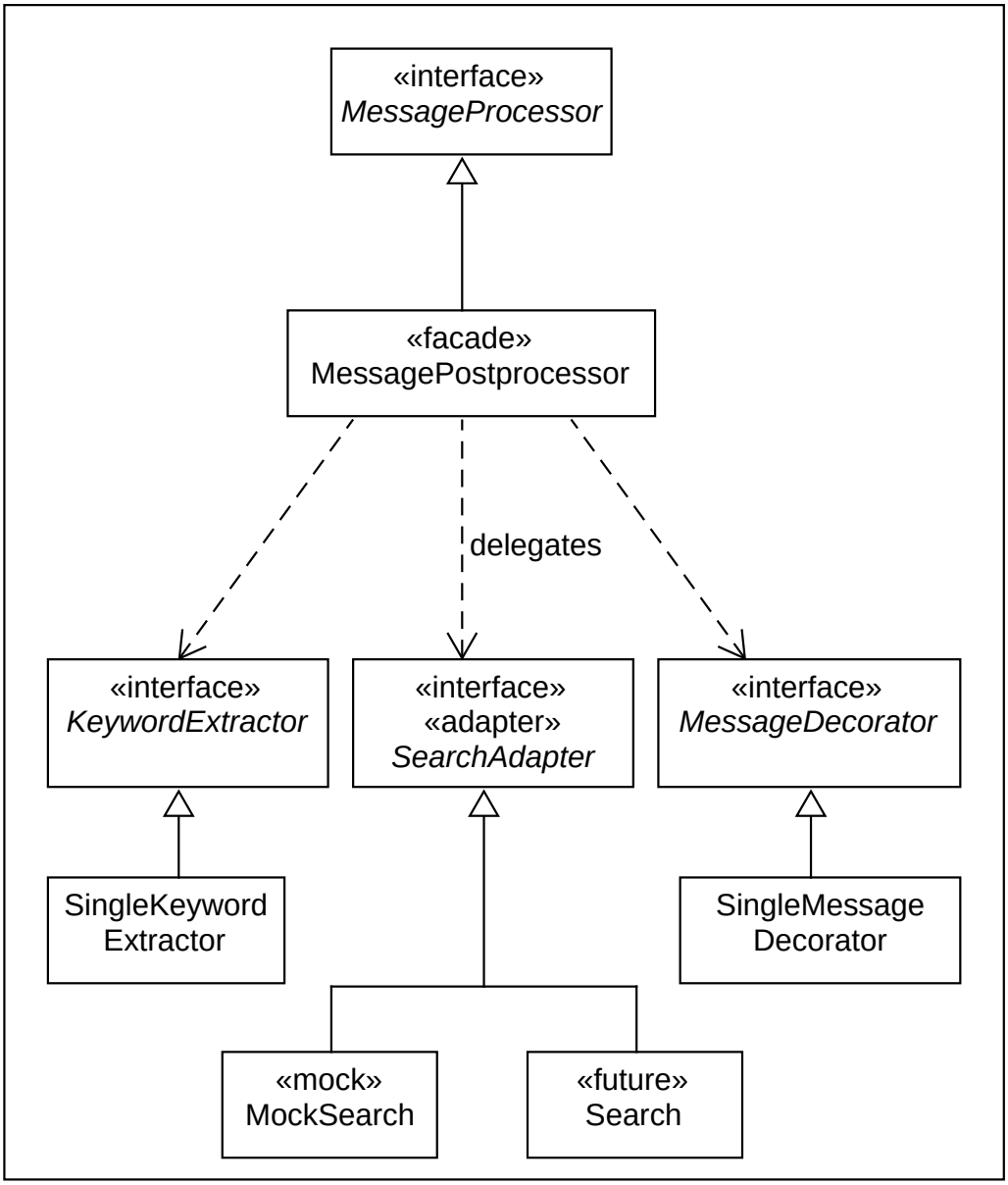

Figure 4.5: Postprocessor class diagram

core chatbot system's expectations by adapting the expectations of the system on the other side appropriately, in line with the "plugin model" architecture (Gamma et al, ibid, pp.139-150; Grenning, 2009, p.119; Figure 4.1).

This concludes the explanation of the main message system exchange pipeline. This model allows us to keep the external tools "at arms length", as opposed to tightly coupling with it and making it almost impossible to replace it in the future. The design also afforded us an intuitive way to overcome the limitations of RiveScript and to integrate with an external system. Finally, it exposes a simple API to the caller.

4.3 The Brain and Data Models

We will now discuss how the brain interfaces with the conversation management logic, and then briefly discuss how user concerns are modelled before moving on to discussing how messages and conversations are modelled (Figure 4.2).

4.3.1 The Brain

The brain folder contains the RiveScript language files. These define the matchable patterns and reply templates and the topics structure as seen from the brain. Topics limit the scope of pattern matchers depending on the the stage of the conversation between user and system, allowing only responses pertinent to the topic to be returned to the user.

The begin.rive, the global.rive, and the python.rive files play each a special role: begin.rive defines basic grammar substitutions and basic synonyms as arrays. The

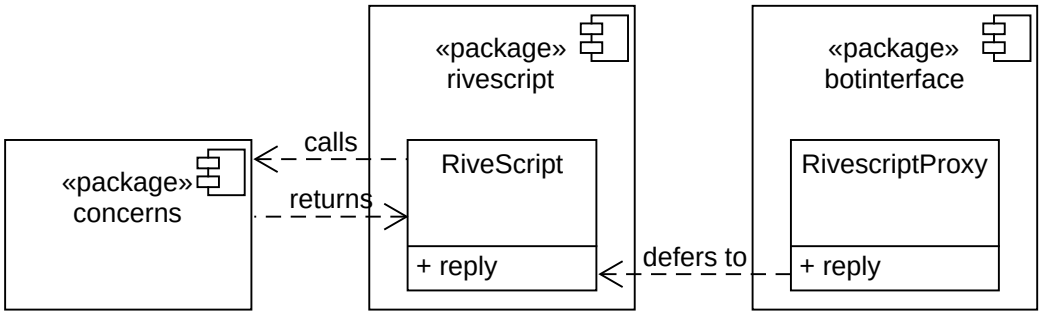

Figure 4.6: Conversation management diagram

global.rive file defines the global scope, that the other topics inherit from. Finally, the python.rive file defines the Python macros that can be called from within RiveScript conditionals and response templates.

4.3.2 Conversation Management and Concern Models

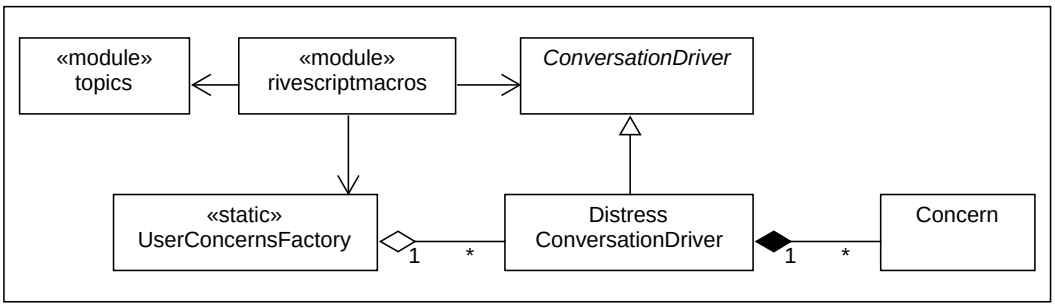

Figure 4.7: Concerns class diagram

The concerns package define data models to represent the user concerns so that these can be be queried by the brain to decide how to move the conversation forward. The conversation concern logic is accessed through Python object macros

written into the python.rive brain file (see above).

The Python statements in these macros import the rivescriptmacros module (Figure 4.6). This module communicates with the ConversationDriver interface, which decides what topic to talk about next, and whether a query via the search engine should be made (Figure 4.7). This is accessed through a static UserConcernsFactory module that lazily instantiates a ConversationDriver for each different userid. Each ConversationDriver instance possesses a record of concerns for the user that it uses to make decisions[2].

Finally, the topics package contains hard-coded information about the relation of microtopics to macrotopics, inferred from the CC (see Chapter 2). This is used to control the internal state of the brain with respect to conversations with users, changing the available set of pattern matchers based on the macrotopic being discussed.

4.3.3 Conversation and Message Models

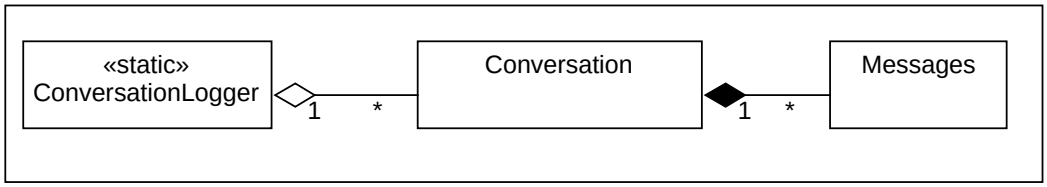

Figure 4.8: Conversation class diagram

The messagelog package defines very simple message logging facilities, primarily for use in the future to be stored durably (Figure 4.1; Figure 4.8).

2. Concretely, in the implementation provided, these are DistressConversationDriver instances that keep track of what the user concerns are and their priority order based on distress scores.

Additionally, the current implementation of the chatbot requires the argument to the reply method of the BotRivescript class to be a Message instance. This is so that the original single argument interface is preserved while both userId and message content (as required by the underlying RiveScript object) can be forwarded. This is also so to leave it the caller responsibility to decide userIds from outside the system, to be able to keep track of the data models. The user ids, in fact, are used to retrieve the relevant data models (conversation logs, concerns) for all users.

4.4 Machine Learning

Finally, we look at the parts of the system that leverage machine learning (Figure 4.2). These are the categorizer and the synonym generator discussed in Chapter 2.

4.4.1 Categorization

The categorization module does not, for the most part, define classes. It exposes essentially procedural code, and leverages the facilities of an NLP library that is built on top of the NLTK: TextBlob (Loria et al, 2016). The reason why this other package was used for the categorization component instead of just the NLTK is primarily its capacity to easily deserialize a classifier and the facilities it exposes to classify several distinct sentences within a single String, which could prove useful when dealing with a single user input that consists of a sequence of distinct sentences[3].

The package provides a way to train, serialize, deserialize and evaluate a single label classifier, but no easy way to swap the concrete classifier instance and no high level

3. https://textblob.readthedocs.io/en/dev/classifiers.html#classifying-textblobs

abstraction in the form of an interface or a façade. This package is the least well designed element of the system: this was the product of the author learning Python, becoming more comfortable with the applicaiton of software design principles while working on the system at the same time.

The reason this subpackage was not used in the final pipeline implementation is that it took too long for the survey that was created to collect relevant data to produce results. But it is possible in principle to plug a trained categorizer at the preprocessor level of the message pipeline in order to add a tag to the user message, in order for this to be used by the chatbot framework to provide better replies to the user, as originally intented (see Chapter 2; and Figure 4.4).

4.4.2 Synonym Generation

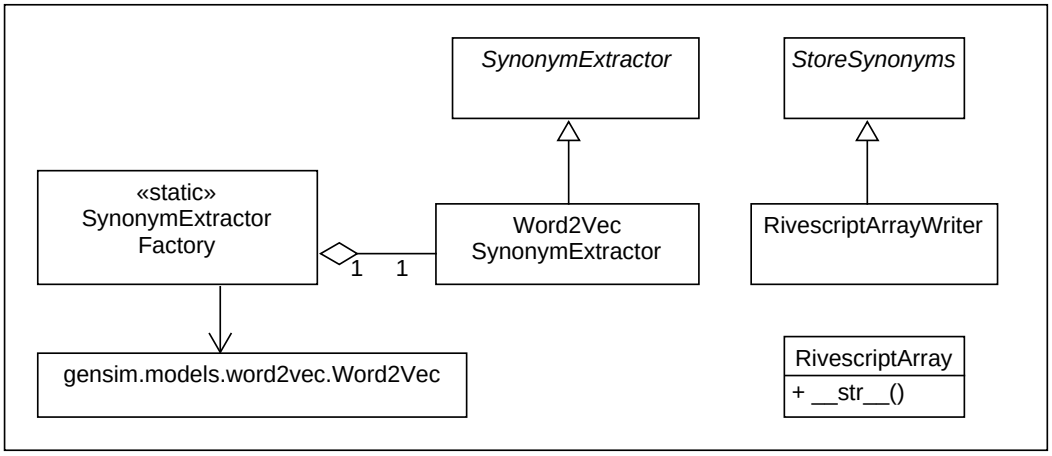

Figure 4.9: Synonym generator class diagram

The final package examined is the synonym generator (Figure 4.9). The use case for this package is to aid in the construction of a RiveScript brain by avoiding

having to manually define common synonyms to simplify the pattern matchers in the brain[4].

The SynonymExtractorFactory lazily instantiates a single pre-trained word2vec language model through the use of the gensim library. This can then be asked to produce synonyms for a given word. The synonyms produced are then used to produce RiveScript arrays.

These are represented in Python by the RivescriptArray class, the ___str___() method (equivalent to Java's toString()) of which is used by the RivescriptArray-Writer to write syntactically correct RiveScript arrays to file[5].

4.5 Conclusion

This concludes the overview of the system design and implementation. We have looked at the general design principles adopted for the project, the high level abstractions provided, then the details of the particular subsystems in the implementation provided.

4. For example, in the following pattern we refer to arrays in order to capture semantically equivalent ways to express the desire to change topic: + [*] (@desire) (@discuss) (@other) [@topic]

5. The RiveScript framework offer ways to stream RiveScript code into a running interpreter, see: http://rivescript.readthedocs.io/en/latest/rivescript.html#rivescript.rivescript.RiveScript.stream.

Chapter 5

System Testing and Evaluation

> "The act of writing a unit test is more an act of design than of verification."

Martin, 2003

This chapter describes the process followed to produce the suite of tests that verify the functioning of the system at the level of core logic and the higher level system behaviour. This chapter also describes the process of classifier evaluation provided by the classification package.

The full test coverage results and categorizer evaluation listings are found in Appendices B and C.

5.1 Testing Strategy

Given the topic of the project, the testing strategy adopted focuses on verifying the behaviour of the individual units of code and the higher level behaviours of the system. This is accomplished through systematic testing with a test-first approach: the test defining the specification for the next system increment is constructed before the unit that will have to pass it (Beck and Andres, 2004, pp.50-51, 97-102; McConnell, 2004, pp.499-).

The integration tests also verify the behaviour of the chatbot brain, where messages are sent to the bot brain and the test fails if the reply returned does not fall within a predefined range of acceptable replies. For example:

```
#tests/tests_integration/test_botinterface/test_preprocessedpractical.py
def test_work():
    resetpractical()
    # perform:
    messages = ["I am underperforming at work because of stress",
    "Something about work ...", "I am afraid to lose my job"]

    for msg in messages[:]:
        reply = bot.reply(Message(USERID, msg))
        # test:
        found = False
        good_replies = ["Are you afraid for your job security?",
        "Would you say this is also a money concern?",
        "How has your condition been affecting your work?"]
```

```
        for good_reply in good_replies:
            if good_reply == reply:
                found = True
                break
        assert found, reply
```

This test first resets the conversation to the "practical" topic in the RiveScript brain, then sends topic pertinent user chat messages to the bot for processing. The both reply is tested at each message against a set of pertinent replies (which represent the replies that have been coded into the chatbot brain for that particular type of topic). This and other tests like it are run to verify the behaviour of the chatbot at the brain level. Furthermore, other tests verify that the chatbot brain changes its state based on the topics that have been discussed throughout the conversation, and other criteria (such as whether it is appropriate to make a query through the search engine). All units and integration tests were made to pass.

One other important form of testing is acceptance testing. The lack of this sort of testing from the project is due to the tight schedule of project delivery and the fact that while there were initial hopes of getting cancer patients to try out the software these were later dismissed as concerns were raised about getting access to the general public via the UCL Hospital.

5.2 Test Driven Development

The test-first strategy for testing the core logic of the project is TDD (Martin, 2009, pp.122-123)[1]. There are three main benefits associated with this practice.

1. TDD is not free of controversy, see Hansson (2014) and Fowler et al (2014).

The first is that it encourages to take on the perspective of the caller of the code, before any code has been written which makes the code easy to call and use to the programmer who is going to consume it. Secondly, it encourages a loosely coupled design. Testing the BotRivescript class in isolation, for example, requires that its associated objects be replaced with mocks providing very predictable behaviour, so that the logic of the class can be scrutinized without external interferences.

Third, it leads to self-documenting code. The tests describe how to use the system and as long as the tests are continuously made to pass, they will never be out of date as comments and Javadoc are always at risk of becoming[2].

For example, the following function of the messagelog package integration test demonstrates how to use both the chatbot and message logging facilities provided by the present package in a way that is technically accurate and intuitive enough that programmers should be able to understand how to use it without further guidance:

```
def _sendUserMessageAndLog(userid, message):
    ConversationLogger.logUserMessage(message)
    reply = bot.reply(message)
    assert reply is not None
    ConversationLogger.logSystemReplyForUser(reply, userid)
    return reply
```

The employement of TDD enabled the tests for the system to achieve 95% code coverage. See Appendix B for test coverage listings and more details.

2. Not all tests strictly adhere to the rules of TDD because the author was learning the discipline while working on the project. As a result, the later a unit was designed, the better the tests for it were.

5.3 Categorization Evaluation Strategy

A classifier is evaluated by looking at certain metrics of its perfomance during testing. A very simple metric is accuracy: the number of correctly labelled data points over the total number of test cases (Bird et al, 2014, Section 3.2).

The way the categorization package provides an evaluation of the classifier is through the ClassifierEvaluator class. The interface of this class allows access to accuracy, precision, recall and F metrics for the given classifier over a "gold" standard data set (Perkins, 2010; Sebastiani, 2002, pp.32-37). Perhaps the most useful of these is the F measure, as Sebastiani (ibid, p.34) warns that accuracy tends to be maximized by the "trivial rejector" or the classifier that tends to assign no label to all data points.

It is important to note, however, that the precision and recall the evaluation module provides for a classifier are individual to the particular label being tested for, and either macro- or micro-averaging should be used to extract global classifier effectivness metrics (Sebastiani, ibid p.33; Yang and Liu, 1999, p.43).

The type of data we wanted our classifier to produce a label for was a short chat message. The label had to be in the specified range of categories, in order to better inform the chatbot with the capacity for generalization of a machine learning approach. There is no corpus of pairs of message - label of the relevant kind. Therefore, the author created a survey which Dr Ramachandran asked the team to complete[3]. This resulted in 222 data points.

Because of the fact that all of the data extracted via the questionnaire has a balanced number of expected labels for categories, there is no concern about the

3. The survey can be found here: https://goo.gl/forms/ylkI50XckV9yvCCm2.

training data being skewed with low positive cases for any one category, therefore micro-averaging is probably the best overall measure, although both are included in the evaluation results.

The results themselves for the categorizer implementations provided by the NLTK are disappointing, primarily because of lack of problem-specific feature selection and possibly idiosyncrasies in the test set. See Appendix C for full results listings and further discussion.

Chapter 6

Conclusion

> When are you done? Since design is open-ended, the most common answer to that question is "When you're out of time."
>
> **McConnell, 2004**

This chapter reviews the project goals and aims, finally, the author expresses his thoughts on the project going forward.

6.1 Project Goals Review

The goals and aims introduced in Chapter 1 are here reviewd. All references to "RQ" below refer to the requirements table in Chapter 3.

- **Design and implement a chatbot architecture tailored to the issues surrounding software systems in healthcare (in particular around treatment of sensitive patient data)**

As discussed in Chapter 2, there are crucial compliance concerns around the sensitivity of patient data that have driven the implementation decision not to employ third party APIs for the processing of natural language user input, even where this may have significantly simplified the chatbot brain implementation task (see RQNF0, RQ1). From this it is possible to see the project achieved all key requirements and several additional requirements[1].

- **To integrate with a specialized search engine (developed by another member of the team)**

While development over the current project had terminated before the search engine was completed, a clear reference on how to integrate between the systems had been provided (and materialized in the integration testing "MockSearch" class and shown integrated at the postprocessing stage; see also RQ5).

- **To explore other applications of NLP that could be useful to extract information from natural language data.**

The current project has used NLP to investigate a hybrid approach to chatbot technology, making use of both rule or grammar based technologies and machine learning with the categorizer (RQ0, RQ2, RQ3). Secondly, NLP was used for the synonym generation (RQ8).

- **To implement a chatbot brain using open source technology.** As mentioned, a chatbot brain was implemented in RiveScript, after reviewing the available choice of technologies given the restrictions on patient data

1. The requirements not covered by this list were not selected as strictly necessary for the project success (*May* under MoSCoW: RQ8, RQ9). RQ7 and RQ8 echo the initial uncertainty with respect to the technologies the other team members were going to use, later the whole team used Python making such concerns redundant, however they are left in the list as they would need to be considered should the technologies around the chatbot component change in the future.

(RQNF0, RQ1).

- **To develop the system with Macmillan eHNA as the main reference.**

As discussed in Chapter 2, it was with reference to the CC used in one of Macmillan eHNA forms that was used for the concrete implementation of the topics in the "concerns" subpackage (RQNF1).

6.2 Personal Aims Review

- **Learning Python in an effort to gain exposure to a new programming language**

The Python programming language was used to implement the majority of the project with the exception of the specialized RiveScript language used to define the pattern matchers in the chatbot brain.

- **Leverage the author's background in computational linguistics, and explore the field of natural language processing**

As mentioned in the previous section, the author investigated the use of NLP to text classification and synonym generation, in addition to significantly more reading in NLP than resulted useful for the project.

- **Learn about applications of machine learning to natural language processing**

The categorization component involved the development of supervised learning language models, while the synonym generation made use of a pre-trained unsupervised

learning model.

- **Improve software engineering skills by applying best agile methodology practices**

A significant amount of reading into agile methodology practices was demonstrated in the project in the system design, implementation and testing.

6.3 Future Work

The primary goal of the project was to lay the ground work, architecture and research for further iterations of the project. As emphasized throughout Chapter 4, the system was designed for extension and modification. The coverage of unit and integration tests provides confidence to any programmer continuing the work here started that the system still displays all behaviours it did when it was delivered at the end of this project, no matter what changes were made during revision and expansion.

The "plugin" model makes the system independent of the IO device that delivers it (in the case of the group project, a webserver) which may be in the future a mobile app, a CLI client, or anything else. It is, in fact, the caller's responsibility to import this project's packages and modules, and to use them as documented in the test cases. Similarly, the data models gathering conversation and user concerns data should be used in a similar fashion: independently of the durable storage solution adopted (whether this is SQL, NoSQL, host filesystem or any other) it should be the caller's responsibility to import this project to serialize the data[2].

2. Although both consumers of the system would benefit from the creation of a unified façade instead of having dependencies across multiple subpackages of the system (Gamma et al, 1995, Chapter 4.5).

There are various ways in which the project could be extended. First of all, the author advises exploring an open source framework different from RiveScript given the limitations discovered within it during this project (or if the client believes it appropriate using third party remote APIs.). See Appendix A for instructions on how to replace RiveScript with some other alternative, and see Chapter 2 for a discussion of alternatives.

Secondly, the categorizer module (to which only a week was dedicated during this project) could be made more robust and extended to use sequence classifiers, potentially extending the NLTK open source project (see Appendix D for initial research in this direction). Eventually, this should become part of the preprocessing layer to inform the input to the rule-based chatbot enigne with the output of a machine learning component.

Third, generation of RiveScript (or other specialized languages) could be investigated following up on the initial results with the synonym generation package. This could dynamically generate synonyms to help improve pattern matchers.

Finally and most importantly, it may be desirable to investigate information-retrieval techniques to extract information from the conversation data gathered during conversations. One of the biggest obstacles to providing cancer patients with better care is the time it takes to create a care plan. Through the extraction of information about the patient's concerns ahead of time, it may be possible to shorten the duration of the care plan appointments, and therefore allow a larger number of patients to be provided with a care plan. Of course, it would still be necessary to first gather relevant data through the conversational UI or a different solution.

Appendix A

System Manual

As stated before, the scope of this report is limited to the core chatbot system which interfaces with a webserver for delivery to a user in the larger team effor. For this reason, no user manual is provided.

A.1 Source Code

The full project source code is available at: https://github.com/nterreri/peach-bot. It can also be found under the PEACH chatbot team project repository under the '/FlaskWebProject/chatbot/' directory therein, together with the code produced by the rest of the PEACH chatbot team (https://github.com/andreallorerung/peach-chatbot-alpha/tree/master/FlaskWebProject/chatbot).

In order to obtain the code it is possible to download the repository as a zip file from the relevant github.com webpage, or use the git software to clone into the repository (*$ git clone https://github.com/andreallorerung/peach-chatbot-*

alpha). The categorizer package is provided as a git submodule to the chatbot package. Cloning or downloading the repository will not automatically download the submodule content. To download all submodules when cloning, use *$ git clone –recursive https://github.com/andreallorerung/peach-chatbot-alpha*. It is not possible to do the same if downloading the repository as a zip file, which means it is necessary to navigate manually to the categorizer repository (https://github.com/nterreri/peach-categorize) before downloading it[1].

The project is meant to be compiled through a Python 2.7 interpreter, no support is provided for Python 3.x. This is primarily because of the preference of members of PEACH. The project has been only compiled and tested using the default C Python compiler which should come by default with any official Python distribution.

A.1.1 Dependencies

The project uses the following Python packages:

```
gensim==0.13.1
nltk==3.2.1
pytest==2.9.2
pytest-cov==2.3.1
rivescript==1.14.1
textblob==0.11.1
```

These in turn depend on should be automatically installed when installing them via the *pip* utility[2]. The most straightforward way to install the dependencies is to

1. When wanting to download the submodule after having cloned into the repository non-recursively, use *$ git submodule update –init –recursive* to download the submodule.
2. https://en.wikipedia.org/wiki/Pip_(package_manager) Python Software Foundation, 2016

use *pip install -r FlaskWebProject/chatbot/requirements.txt*, but the dependencies may also be installed piecemeal.

A.2 Package Structure

The chatbot Python package contains several subpackages (directories) diving up the software in large abstractions:

```
chatbot/
|-- botinterface/
|-- brain/
|-- categorizer/
|-- concerns/
|-- messagelog/
|-- postprocess/
|-- preprocess/
|-- synonym/
`-- tests/
```

A.2.1 The Chatbot

The *botinterface/* package exposes the interface class for the core chatbot component in the *botinterface/bot_interface.py*, and provides an implementation in *botinterface/bot_rivescript.py*. In order to depend only on the API, it is possible to ask the *bot_builder.py* module (located in the project top level directory) to construct a concrete subtype of the interface.

This last module exposes a *build()* method that will return a default instance of the bot, which will assume the RiveScript brain files are located within a *brain/* directory immediately accessible. The same package defines a *BotBuilder* class that the caller can use to customize the chatbot instance being returned. For example, the following code listing will initialize a chatbot with a brain located at the specified filepath ("path/to/brain"):

```
botBuilder = bot_builder.BotBuilder()
botBuilder.addBrain("path/to/brain/")
chatbot = botBuilder.build()
```

The RiveScript-based chatbot instance returned expects messages to be forwarded to it in the form of a *messagelog.message.Message* instance. The Message class simply requires that both a userid and message content be provided. This is to enforce conformity with the interface defined in the *botinterface/bot_abstract.py* module: the *reply(message)* method should take a single argument while the caller retains control of the creation and management of userids independently of the core chatbot component (Figure 4.1).

As mentioned before, the conversation logging facilities provided by the *messagelog* package are not automatically used by the chatbot instance, instead something like code listing (#SOMENUMBER) is necessary.

Why RiveScript should be removed

Overall, RiveScript has been a really useful tool to rapidly set up and start programming a chatbot brain. Unfortunately, many shortcomings were found with it as the project evolved.

The first thing to take into account is that the current implementation of the chatbot brain is incomplete and not very articulate. It is incomplete because it is at the moment impossible to move on from any of the topics other than the phyisical issues topic, therefore any user who has concerns other than physical will never trigger the macro that changes the state of the DistressConversationDriver data model for the user to move on to the next topic. It is not articulate because very little time was spent working on the brain implementation, given the focus system architecture and design.

The hand written rules that frameworks like RiveScript (and competitors as described in Chapter 2) need take a significant amount of time to write and test, more than common programming code. Secondly, numerous issues were met during development, including textbook use of RiveScript language syntax constructs causing uncaught exceptions to be thrown from within the external framework , and issues with inconsistent internal state with the framework uservariables that took three days (amounting to 10% of project core development time) to debug and fix, due to poorly documented behaviour when changing and using the same user variable in the breath of the same pattern matcher (including redirects).

Finally, the Python statements in the RiveScript object macros cannot be extracted to a separate file. This is because the "rs" reference available therein (which is the running RiveScript interpreter instance) refuses to be called outside of a .rive file, and will fail with an exception if an attempt is made to do so. To call this object's properties is necessary to gather information such as the current user session, and the current state of the user variables. This makes the statements in the macros untestable.

Due to these shortcomings, the recommendation going forward is to consider using a different chatbot framework. As mentioned, only a minimal brain implementation

has been provided with the architecture and it would not be an irreparable loss to start with this aspect of the system from scratch. The reader is redirected to the discussion of alternatives in Chapter 2.

How to Replace RiveScript

In order to replace the RiveScript interpreter, it is necessary to create another implementation of the Interpreter class, adapting the implementation provided in RivescriptProxy. Depending on the framework used, it may be necessary to create a proxy for a unit of software implemented in other languages than Python. This is made easier by the various Python compilers that allow other languages to be used together with Python[3].

Effectively, the BotRivescript class only expects its interpreter to expose a "reply" method, but other aspects of the implementation may change as well such as how to start the conversation with the user, all changes necessary should however be confined to the new proxy (see also Figure 4.3).

Requirements for the pre- and postprocessing of messages are also likely to change, therefore similar considerations apply to their packages. Finally, it would be necessary to alter the *bot_builder* module to include the new or modified implementation, as this would need to know and name (import) the new implementations. See Chapter 2 for a discussion of alternatives to RiveScript.

3. See Reitz and Schlusser, 2016 here: http://docs.python-guide.org/en/latest/starting/which-python/

A.2.2 Categorizer

The categorizer can be used through the *demo.py* module. Effectively, this module should in the future become a more complete utility callable from the command line, that would train or load a serialized classifier before running it against a development or test set. As it stands, running the module will always result in the classifier being trained, run against the dev set and finally tested automatically.

A.2.3 Synonym Generation

The synonym generation facilities are meant to be accessed through the *synonym/synonym_extractor_factory.py* module. This module lazily instantiates an instance of the *Word2VecSynonymExtractor* class (defined in *synonym/synonym_word2vecextractor.py*).

Instantiating this object requires will take up around 6-8 GBs of main memory, due to the size of the pretrained data model it uses. After it has been loaded, it is possible to obtain synonyms from it given a word. Doing this may cause a lot of pages to be swapped in and out of virtual memory as the model retrieves similar words based on cosin distances. Sample usage of this module (adapted from */tests/tests_integration/test_synonym/test_synonymmodelfactory.py*):

```
synonymsFor = dict()
for word in WORDS_TO_GET_SYNONYMS_FOR:
      synonymsExtracted = extractor.extractSynonyms(word)

      synonymsFor[word] = synonymsExtracted
```

This will return a list of words which the model believes are synonyms of the argument to the *extractSynonyms()* method. As described in Appendix C, this particular implementation was underperforming, but provides the blueprint for future better performing implementations (see Chapter 2 for a discussion of alternatives).

A.3 Full Concerns Checlist

As mentioned in Chapter 2, the full concerns checklist is presented here for reference, these are encoded in the *concerns/topics.py* module:

```
Physical concerns 1          Physical concerns 2
- Breathing difficulties     - Sore or dry mouth
- Passing urine              - Nausea or vomiting
- Constipation               - Sleep problems/nightmares
- Diarrhoea                  - Tired/exhausted or fatigued
- Eating or appetite         - Swollen tummy or limb
- Indigestion                - High temperature or fever

Physical concerns 3          Physical concerns 4
- Getting around (walking)   - Changes in weight
- Tingling in hands/feet     - Memory or concentration
- Pain                       - Taste/sight/hearing
- Hot flushes/sweating       - Speech problems
- Dry, itchy or sore skin    - My appearance
- Wound care after surgery   - Sexuality

Practical concerns 1         Practical concerns 2
```

- Caring responsibilities
- Work and education
- Money or housing
- Insurance and travel
- Transport or parking
- Contact/communication with NHS staff

- Housework or shopping
- Washing and dressing
- Preparing meals/drinks

Family concerns

- Partners
- Children
- Other relatives/friends

Emotional concerns 1

- Difficulty making plans
- Loss of interest/activities
- Unable to express feelings
- Anger or frustration
- Guilt
- Hopelessness

Emotional concerns 2

- Loneliness or isolation
- Sadness or depression
- Worry, fear or anxiety

Spiritual or religious concerns

- Loss of faith or other spiritual concerns
- Loss of meaning or purpose of life
- Not being at peace with or feeling regret about the past

Appendix B

Unit Tests Results

This appendix reports the unit testing code coverage for all modules in the project. The testing framework of choice is py.test.

B.1 Running py.test

The tests included are meant to be run using py.test (Krekel, 2016). In order to allow the required project packages to be imported, this should be run from the project top level directory (the one overseeing all the subpackages and test folder). The directory or test file to run can be specified as an argument to the Python CLI interpreter:

```
python -m py.test tests/
```

Running this command will run all the tests in the package, which on a fairly powerful machine can take up to 5 minutes, but can potentially make a slower machine hang as for an unreasonable amount of time. This is chiefly due to

the synonym generation tests which require a very large amount of RAM to run efficiently (see above).

Running only the unit tests should take less than a minute[1]. The unit tests, in fact, are designed for quick execution (see Martin, 2009, p.132). Finally, if the pytest-cov package is been installed (Schlaich, 2016) then it is possible to get reports for code coverage about individual subpackages by providing the desired subpackage name as an argument in the following manner:

```
python -m py.test --cov=botinterface tests/tests_unit/
```

This will provide an executable statement test coverage report. The reader is redirected to Chapter 5 for more information about the testing strategy.

B.2 Code Coverage Report

These reports are taken from the pytest CLI because pytest-cov can only write HTML, XML or annotated source code to file, but not plain text as shown here. All tests were, of course, made to pass. Test coverage data follows[2]:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                        Stmts   Miss  Cover
-------------------------------------------------------------
botinterface/__init__.py                        0      0   100%
```

1. It is good to notice at this point the difference in runtime between the decoupled unit tests and the integration tests in this case, running all unit tests for the project takes seconds.

2. Legend: “Stmts” stands for total number of executable statements found in module, “Miss” for statements not executed by any test, “Cover” is the percentage of statements covered by the tests.

botinterface/bot_abstract.py	7	3	57%
botinterface/bot_builder.py	23	5	78%
botinterface/bot_rivescript.py	30	1	97%
botinterface/message_processor.py	5	2	60%
botinterface/postprocessor.py	21	0	100%
botinterface/postprocessor_example.py	20	0	100%
botinterface/preprocessor.py	20	0	100%
botinterface/rivescript_loader.py	13	1	92%
categorize/__init__.py	0	0	100%
categorize/classifierDeserializer.py	5	0	100%
categorize/classifierSerializer.py	4	0	100%
categorize/dataset_reading.py	12	4	67%
categorize/dataset_splitting.py	10	0	100%
categorize/develop.py	10	10	0%
categorize/evaluation.py	46	2	96%
categorize/training.py	4	0	100%
concerns/__init__.py	0	0	100%
concerns/concern.py	10	0	100%
concerns/concern_factory.py	13	0	100%
concerns/drive_conversation.py	54	0	100%
concerns/drive_conversation_abstract.py	11	5	55%
concerns/equivalence.py	32	1	97%
concerns/rivescriptmacros.py	24	3	88%
concerns/topics.py	14	0	100%
messagelog/__init__.py	0	0	100%
messagelog/conversation.py	6	0	100%
messagelog/conversation_logging.py	21	0	100%

messagelog/message.py	8	0	100%
postprocess/__init__.py	0	0	100%
postprocess/keyword_extractor.py	5	2	60%
postprocess/keyword_extractor_single.py	30	0	100%
postprocess/message_decorator.py	5	2	60%
postprocess/message_decorator_single.py	29	0	100%
postprocess/postprocessor_builder.py	5	0	100%
postprocess/search_adapter.py	5	2	60%
preprocess/__init__.py	0	0	100%
preprocess/preprocessor_builder.py	6	0	100%
preprocess/stemmer_factory.py	3	0	100%
preprocess/stemming.py	5	2	60%
preprocess/stemming_lancaster.py	7	1	86%
preprocess/stopword_remover_nltk.py	6	0	100%
preprocess/stopwords_remover.py	17	1	94%
preprocess/stopwords_remover_lenient.py	5	0	100%
preprocess/tokenizer.py	5	2	60%
preprocess/tokenizer_simple.py	7	1	86%
synonym/__init__.py	0	0	100%
synonym/rivescript_array.py	9	0	100%
synonym/rivescript_writer.py	7	7	0%
synonym/store_synonyms.py	4	4	0%
synonym/synonym_extractor.py	5	2	60%
synonym/synonym_extractor_factory.py	13	5	62%
synonym/synonym_word2vecextractor.py	17	0	100%
TOTAL	527	52	89%

Note that there are several abstract classes that are also counted in the total executable statements even though they should not, for example *MessageProcessor*:

```
#botinterface/message_processor.py
class MessageProcessor(object):
    '''Interface a message content processor is expected to implement'''
    def __init__(self):
        raise NotImplementedError("MessageProcessor is an interface")

    def process(self, sentence):
        '''To preprocess the sentence (content of a message)'''
        raise NotImplementedError("MessageProcessor is an interface")
```

The two "raise" statements are counted when they should not be, similarly with many other classes. This means that the actual test coverage is slightly higher. There are 27 statements from interfaces[3] that should not be counted as statements at all, bringing the metrics to 500 total statements and 25 missed statements bringing coverage to 95%.

There are also integration tests testing the interaction of two or more independent units, or other aspects of the system that do not belong to unit testing, such as using the actual word2vec model instead of a mock. Again, all such tests were made to pass. As mentioned in Chapter 5, the RiveScript brain logic is also extensively tested via integration tests.

Finally, the RiveScript Python macro code within *brain/python.rive* is not consid-

3. Specifically, the interfaces: *botinterface/bot_abstract.py*, *botinterface/message_processor.py*, *concerns/drive_conversation_abstract.py*, *postprocess/keyword_extractor.py*, *postprocess/message_decorator.py*, *postprocess/search_adapter.py*, *preprocess/stemming.py*, *preprocess/tokenizer.py*, *synonym/store_synonyms.py*, *synonym/synonym_extractor.py*, finally the constructor of the abstract class *preprocess/stopwords_remover.py* should also not be counted.

ered and not directly covered by any tests and is in fact impossible to test in isolation. This is because the RiveScript object instance "rs" accessible within these macros cannot be called from inside separate and isolated methods (as it was the author's original intention) and will raise an error if it is passed as a variable to an externally defined unit of Pyhon code. This aspect of RiveScript is poorly documented, and another reason to move away from the framework in the future (Petherbridge, 2009, https://www.rivescript.com/wd/RiveScript#OBJECT-MACROS).

Appendix C

Machine Learning Evaluation

In this section, detailed evaluation results are provided for the categorization task, and a summary evaluation is provided for the synonym generation task.

C.1 Categorizer Evaluation

The survey gathered 222 data points in total (see Chapter 5). The same split was used for the training and testing of both classifier implementations used. The single-label categorizers used in the project are provided by the nltk through the TextBlob package: a Naive Bayes classifier and a Maximum Entropy classifier.

The data splitting logic in the *categorize* package by default simply takes 10% of the labelled data set for training, and another 10% for developement:

```
#categorize/dataset_splitting.py
def split(labelled_data, testsizepercent = 0.1, devsizepercent = 0.1, \
          shuffle = False):
```

The results of running the Naive Bayes classifier against the test set (around 22 test cases) is disappointing:

```
Accuracy: 0.36
Macro averaged recall: 0.29
Macro averaged precision: 0.35
Micro averaged recall: 0.36
Micro averaged precision: 0.36

Recall for 'pract': 0.0          Recall for 'phys': 0.67
Precision for 'pract': 0.0       Precision for 'phys': 0.33
F for 'pract': 0                 F for 'phys': 0.44
Recall for 'family': 0.4         Recall for 'emotion': 0.4
Precision for 'family': 1.0      Precision for 'emotion': 0.4
F for 'family': 0.57             F for 'emotion': 0.4
Recall for 'spiritual': 0.0
Precision for 'spiritual': 0.0
F for 'spiritual': 0
```

All metrics for the 'pract' and 'spiritual' labels are 0 which means the classifier systematically gets them wrong over the data set. The micro and macro averages of precision and recall for all labels score around 33%.

The MaxEntClassifier, instead, scores over the same data split (10% test, 10% dev, 80% train) when made to iterate over the dev set for 10 iterations:

```
Accuracy: 0.55
Macro averaged recall: 0.4467
Macro averaged precision: 0.4911
Micro averaged recall: 0.5454
Micro averaged precision: 0.5454

Recall for 'pract': 0.4          Recall for 'phys': 0.83
Precision for 'pract': 0.4       Precision for 'phys': 0.56
F for 'pract': 0.4               F for 'phys': 0.67
Recall for 'family': 0.4         Recall for 'emotion': 0.6
Precision for 'family': 1.0      Precision for 'emotion': 0.5
F for 'family': 0.57             F for 'emotion': 0.55
Recall for 'spiritual': 0.0
Precision for 'spiritual': None
F for 'spiritual': None
```

Like the Naive Bayes classifier, it scores a 0 in every metric for the 'spiritual' label, but scores better in other global metrics (averaged over all labels).

C.1.1 Comment on Performance

The reason for the performance is most likely the lack of problem-specific feature extraction. The feature extractors that both classifiers use in the current implementation, in fact, is simply that both use the default feature extractor provided

with TextBlob [1].

This feature extractor simply looks at all the words contained within the train set, and encodes features as the occurrence of the word within the document. This means that when the classifier is asked to classify a document it will look simply at what words in the vocabulary seen at training time occur in the it. The purpose of having a development set is exactly to improve our feature extraction, but the timeline and priorities of the current project unfortunately left no time to work on this aspect of the system.

Secondly, given that the data is split in the exact same way for evaluation of the classifier, the failure to correctly label any of the "spiritual" category data points may have to do with idiosyncrasies with the data split, such as no data points of that category occurring in the test set. This has partly to do with the small amount of data available, but could be alleviated using more advanced evaluation techniques, such as k-fold validation (Bird et al, 2014, Chapter 6 section 3.5; Sebastiani, 2002, pp.9-10; Yang, 1999).

Furthermore, see the discussion of sequence classifier for what is perhaps a more interesting direction than simply improving the performance single-label classifers (Appendix D).

1. See http://textblob.readthedocs.io/en/dev/_modules/textblob/classifiers.html#basic_extractor

C.2 Word2Vec Synonym Generation Evaluation

The model used in the implementation is a publically available model that has been trained over a very large data set of Google News data[2].
As mentioned in Chapter 2 and 4, the implementation here for synonym extraction was not used in the system delivered by the team because of poor performance.

As it is possible to see from the integration tests the cosin distance does not always appear to capture semantic synonymity (Mikolov, 2015; McCormick, 2016b). For example, while perfomance may be judged adequate with respect to words like "dad" or "friend", there are cases when it can only associate the word with either misspellings or completely irrelevant terms (such as names of reporters). This is maybe interesting linguistically, but does not help with generating synonyms.

```
#from tests/tests_integration/test_synonym/test_synonymmodelfactory.py
{
    "dad":["Dad","father","grandpa","daddy","mom","stepdad","son","granddad",
            "uncle","brother"]
    ,"hopeless":["utterly_hopeless","forlorn","hopelessly","miserable",
                "pathetic","pitiful","helpless","seemingly_hopeless","bleak",
                "futile"]
    ,"education":["eduction","eduation","LISA_MICHALS_covers",
                    "Matt_Krupnick_covers","educational","educa_tion",
                    "edu_cation","educations","professionals_CEC_SmartBrief",
                    "curriculum"]}
```

As mentioned in the discussion, the implementation of this concrete class was done

2. The publically available model used is not provided with the system, but obtainable from: https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

with the help of the Gensim library (Rehurek, 2014; Rehurek and Sojka, 2010; McCormick, 2016a). The cause for the poor performance with respect to the task at hand is perhaps to be found with the dimension of the vectors the model used was trained with (as a training parameter, see Ellenberg, 2015). The reader is redirected to the discussion of alternatives in Chapter 2.

In general for the project going forward, as long as RiveScript or similar rule-driven chatbot frameworks are used, it may well be possible to similarly generate data for use in similar ways, prior exploration of other alternatives to the particular model used in this implementation.

Appendix D

Advanced Research into Categorization

The classifiers used in the project would be used to give labels to distinct documents, however the documents we are interested in classifying are not independent from each other: a conversation is a rich in context, and no feature of the approach outlined so far even considers this aspect of the problem.

One thing we could do to improve the classifier is to provide a level of uncertainty, where user inputs may fail to be assigned any of the predefined labels. This not an attempt to account for essentially neutral inputs such as "Yes" or "I don't know", that do not contribute to establishing the topic of the conversation. This is rather a way to try and account for indeterminateness of the topic at hand.

If a patient were to mention an issue that scores a reasonable confidence level from the classifier in both emotional and family categories, but both scores are below a certain threshold, then perhaps the best thing for the classifier is to not assign a

label to the input. This accounts for the inherent "fuzziness" of certain topics of discussion.

Something else we could do is consider conversation topics as the states in a finite state machine, and account for varying probabilities to move into different topics, given the current state. Emotional issues may, for example, seem closer to family issues than physical issues. However, this seems quite an ambitious architecture to impose over a simple classifier, which leads us onto the next topic.

D.1 Sequence Classifiers

A sequence classifier is similar to a single or multi-lable classifier, except it considers each data point as a member of a sequence: the data points are assigned labels not independently of each other but as part of one sequence (Bird et al, 2014, Chapter 6 sections 1.6-1.7; Jurafsky and Martin, 2014, p.1). This type of classifier intuitively better suits the task of classifying sequences of utterances in a conversation, than a classifier that only decides a label for the input considered in isolation.

A Markov chain is a weighted finite state automaton (a directed weighted graph with a finite number of nodes), where the weight on all transitions from any state represent the probability of moving to the state the transition points to (Jurafsky and Martin, 2014, p.2). [#PICTUREWOULDBENICE?]

More formally: A set of states: $Q = \{q_1, q_2, ..., q_n\}$ A transition probability matrix: $A = \{a_{01}, a_{02}, ..., a_{n1}, ..., a_{nn}\}$ And a start and an end (accepting) state: q_0, q_F

A Markov chain may already be sufficient for us to do some useful modelling. In particular, we may construct a simple (ergodic, fully connected) directed weighted

graph (with appropriate constraints on the weights to represent a probability distribution from each state) that connects our five states together represeting the macro categories of concerns in the CC. Having manually set probabilities for state transitions, we would use our simple classifier to decide the category for the next user input in isolation, then confront this result against the likelihood of a transition in that direction, before finally deciding the label.

We could take the problem one step further: a Hidden Markov Model (HMM) is a generalization of a Markov chain that distinguishes between observed and hidden events. For example, in a part of speech tagging task, the observed is the surface level sequence of words, the hidden is the sequence of tags describing the syntactic structure of the utterance. This distiction would allow, in our case, to distinguish between a sequence of natural language user inputs, and the sequence of conversational topics "hidden" within.

Formally, to the properties of a Markov chain outlined above we add: A sequence of non-hidden observations: $O = \langle o_1, o_2, ...o_m \rangle$ A function describing the probability of observation o_j being generated (called probability of emission) from state i: $b_i : O \mapsto [0, 1]$

At any point during the conversation, we are really interested in determining the probability distribution of the *reachable states* given the sequence of observations and the corresponding sequence of states so far identified, in order to determine what the best *next state* is. Given the next observation, we compute the probability of emission of that observation from all the plausible states reachable from the current state. We then may want to do a harmonized average of these probabilities and the known state transition probabilites, in order to decide the next state (the next label for the user input in our case).

The problem that still needs to be addressed is how we determine the emission probability distribution. The user input will likely vary from session to session: no two sequences of user inputs, representing the half of the conversation the user contributes, are going to be alike. It may be possible to reduce each user input to a set of semantically salient terms, terms that are associated with one or another topic. For example, mention of proper names or nouns denoting family members (such as mother, aunt etc) may be indicative that the topic is family. In contrast, terms associated with emotional problems may be indicative of the topic being emotional issues. In a way, we are returning back to a simple classifier to decide what the observation to be then fed to the sequence classifier should be.

D.2 The limits of the NLTK

The implementation produced by this project does not make use of sequence classification. There are two reasons for this. The first is that, as may have become evident through the preceeding section, the problem is complex enough to warrant exploration in a separate dissertation. And given the focus of the current project, to dedicate more resources to this topic would require neglecting other, perhaps more important aspects of the problem to be solved.

The second is that the NLTK does not provide implementations of any sequence classifier (at the time of writing). Looking through the public open source repository it is possible to see that an interface sequence classifiers would be expected to conform to within the NTLK package has been written, but is commented out in the code, and no implementation of the interface can be found[1]. There are

1. This link should reference the latest commit to the NLTK project as of 21/08/16: https://github.com/nltk/nltk/blob/991f2cd1e31f7c1ad144589ab4d2c76bee05aa7b/nltk/classify/api.py.

open source implementations of Hidden Markov Models available, but work would be required in order to either fit these to the NTLK system, or to incorporate them within the current project. See for example, "pomegranate" (Schreiber et al, 2016)[2].

2. Here: https://github.com/jmschrei/pomegranate#hidden-markov-models

Appendix E

Code Listing

Partial code listings follow. The full source code can be obtained following the instructions in Appendix A.

References

ALICE A.I. Foundation. *Program AB*, July 2014. Accessed August 21, 2016. https://code.google.com/archive/p/program-ab/.

Apache. *Apache OpenNLP - Welcome to Apache OpenNLP*, July 2015. Accessed August 20, 2016. http://opennlp.apache.org/.

AWS. *Amazon Web Services (AWS) - Cloud Computing Services*, 2016. Accessed August 21, 2016. http://aws.amazon.com/.

Azure. *Microsoft Azure: Cloud Computing Platform and Services*, 2016. Accessed August 21, 2016. https://azure.microsoft.com/en-gb/.

Beall, Jc, and Greg Restall. "Logical Consequence." In *The Stanford Encyclopedia of Philosophy*, Fall 2014, edited by Edward N. Zalta. 2014. Accessed August 18, 2016. http://plato.stanford.edu/archives/fall2014/entries/logical-consequence/.

Beck, Kent, and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 978-0-321-27865-4.

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, et al. *Manifesto for Agile Software Development*, 2001. Accessed August 20, 2016. http://agilemanifesto.org/.

Berger, Allan. *The Rise of the Bots: Bots and the future*, May 2016. Accessed August 17, 2016. https://medium.com/@allanberger/the-rise-of-the-bots-fc3471792019.

Biermann, Alan W. "Fundamental mechanisms in machine learning and inductive inference" [in en]. In *Fundamentals of Artificial Intelligence*, edited by Wolfgang Bibel and Philippe Jorrand, 133–169. Lecture Notes in Computer Science 232. DOI: 10.1007/BFb0022682. Springer Berlin Heidelberg, 1986. ISBN: 978-3-540-16782-2 978-3-540-39875-2, accessed August 18, 2016. http://link.springer.com/chapter/10.1007/BFb0022682.

Bird, Steven, Ewan Klein, and Edward Loper. *NLTK Book*, 2014. Accessed August 20, 2016. http://www.nltk.org/book/.

Brittle, Andrew. "Developing the eHNA." *Mac Voice, the magazine for Macmillan professionals* Winter 2014. Accessed August 17, 2016. http://www.macmillan.org.uk/aboutus/healthandsocialcareprofessionals/newsandupdates/macvoice/winter2014/developingtheehna.aspx.

Brooks, Frederick P. Jr. *The Mythical Man-month: Essays on Software Engineering.* 2 edition. Addison Wesley, August 1995. ISBN: 978-0-201-83595-3.

Dillet, Romain. *Here's the first demo of Viv, the next-generation AI assistant built by Siri creator*, May 2016. Accessed August 17, 2016. http://social.techcrunch.com/2016/05/09/heres-what-viv-looks-like-the-next-generation-ai-assistant-built-by-siri-creator/.

Ding, Xiaowen, Bing Liu, and Philip S. Yu. "A Holistic Lexicon-based Approach to Opinion Mining." In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, 231–240. WSDM '08. New York, NY, USA: ACM, 2008. ISBN: 978-1-59593-927-2, accessed August 20, 2016. doi:10.1145/1341531.1341561. http://doi.acm.org/10.1145/1341531.1341561.

Ellenberg, Jordan S. *Messing around with word2vec*, January 2016. Accessed August 22, 2016. https://quomodocumque.wordpress.com/2016/01/15/messing-around-with-word2vec/.

Ellis, Rob. *Creating a Chat Bot — Free Code Camp*, October 2014. Accessed August 22, 2016. https://medium.freecodecamp.com/creating-a-chat-bot-42861e6a2acd#.o6oloksxo.

———. *SuperScript*, August 2016. Accessed September 4, 2016. https://github.com/superscriptjs/superscript.

Fellbaum, Christiane. "WordNet and wordnets." In *Encyclopedia of Language and Linguistics, Second Edition*, edited by Brown and Keith, 665–670. Oxford: Elsevier, 2005. Accessed August 21, 2016. https://wordnet.princeton.edu/.

Finextra Research. *Rise of the bots: DBS to put virtual assistant in Facebook Messenger*, August 2016. Accessed August 17, 2016. https://www.finextra.com/newsarticle/29318/rise-of-the-bots-dbs-to-put-virtual-assistant-in-facebook-messenger.

Fowler, Martin. *TechnicalDebt*, October 2003. Accessed August 22, 2016. http://martinfowler.com/bliki/TechnicalDebt.html.

French, Jade. *Are you ready for the rise of the bots?*, 2016. Accessed August 17, 2016. http://www.weareamplify.com/presents/are-you-ready-rise-bots/.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 978-0-201-63361-0.

Grenning, James. "Boundaries." In *Clean Code: A Handbook of Agile Software Craftsmanship*, 113–120. Robert C. Martin Series. Prentice Hall, Pearsons Education, 2009.

Hansson, David Heinemeier. *TDD is dead. Long live testing. (DHH)*, April 2014. Accessed August 24, 2016. http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html.

Hawthorne, James. "Inductive Logic." In *The Stanford Encyclopedia of Philosophy*, Winter 2014, edited by Edward N. Zalta. 2014. Accessed August 18, 2016. http://plato.stanford.edu/archives/win2014/entries/logic-inductive/.

Hume, David. *An Enquiry concerning Human Understanding*, 1777. Accessed August 18, 2016. http://www.davidhume.org/texts/ehu.html.

Iaia, Cosimo. *pyAiml-2.0*, January 2016. Accessed August 21, 2016. https://github.com/cosimoiaia/pyAiml-2.0.

Indurkhya, Nitin, and Fred J. Damerau. *Handbook of Natural Language Processing*. 2nd. Chapman & Hall/CRC, 2010. ISBN: 978-1-4200-8592-1.

Is TDD Dead, 2014. Accessed August 24, 2016. http://martinfowler.com/articles/is-tdd-dead/.

Jurafsky, Dan, and James H. Martin. "Hidden Markov Models." In *Speech and Language Processing (3rd edition draft)*, 3rd edition DRAFT. September 2014. Accessed August 21, 2016. https://web.stanford.edu/~jurafsky/slp3/.

Jurafsky, Daniel, and James H. Martin. *Speech and Language Processing, 2nd edition*. 2nd. Prentice Hall Series in Artificial Intelligence. Pearson Education International, 2009.

Krekel, Holger. *py.test*, 2016. Accessed September 4, 2016. http://doc.pytest.org/en/latest/.

Liu, Bing. "Sentiment Analysis and Subjectivity." In *The Handbook of Natural Language Processing*, 2nd, 627–666. Chapman & Hall, 2010.

Loria, Steven. *TextBlob*, June 2016. Accessed August 23, 2016. https://github.com/sloria/TextBlob.

Manning, Christopher, Prabhavan Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Online edition DRAFT. Cambridge University Press, 2009. Accessed August 20, 2016. http://nlp.stanford.edu/IR-book/.

Manning, Christopher, and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*, 1999. Accessed August 18, 2016. http://nlp.stanford.edu/fsnlp/promo/.

Manning, Christopher, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. "The Stanford CoreNLP Natural Language Processing Toolkit." In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 55–60. 2014.

Martin, Robert C. *Agile Software Development Principles, Patterns, and Practices*. Alan Apt Series. Upper Saddle River, NJ: Pearson Education, 2003.

———. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Prentice Hall, Pearsons Education, 2009.

Martin, Robert C. *Screaming Architecture*, September 2011. Accessed August 23, 2016. https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html.

McConnell, Steve. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 978-0-7356-1967-8.

McCormick, Chris. *Google's trained Word2Vec model in Python*, April 2016. Accessed August 22, 2016. http://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/.

———. *Word2Vec Resources*, April 2016. Accessed August 22, 2016. http://mccormickml.com/2016/04/27/word2vec-resources/.

Microsoft Cognitive Services. *Microsoft Cognitive Services*, 2016. Accessed August 17, 2016. https://www.microsoft.com/cognitive-services.

Mikolov, Homas. *word2vec*, January 2015. Accessed August 22, 2016. https://code.google.com/archive/p/word2vec/.

Morton, Dave. *How To Create Your Own Customised Chatbot For Beginners - Chatbots 101*, April 2011. https://www.chatbots.org/ai_zone/viewthread/492/.

Mota, Bruno. *Quick Tip: Install Multiple Versions of Node.js using nvm*, March 2016. Accessed August 22, 2016. https://www.sitepoint.com/quick-tip-multiple-versions-node-nvm/.

NCAT. *Holistic Needs Assessment for people with cancer A practical guide for healthcare professionals*, 2011. http://www.ncsi.org.uk/wp-content/uploads/The_holistic_needs_assessment_for_people_with_cancer_A_practical_Guide_NCAT.pdf.

Node Core Technical Committee and Collaborators. *node*, August 2016. Accessed August 22, 2016. https://github.com/nodejs/node/.

Pandorabots. *chatbots.io*, 2016. Accessed August 17, 2016. https://developer.pandorabots.com/.

PEACH. *Project PEACH*, 2016. Accessed August 17, 2016. https://code4health.org/peach.

Perkins, Jacob. *Text classification for sentiment analysis - precision and recall*, May 2010. Accessed August 24, 2016. http://streamhacker.com/2010/05/17/text-classification-sentiment-analysis-precision-recall/.

Perreau, Elizabeth, and Dave Morton. *Program-O*, May 2014. Accessed August 21, 2016. https://github.com/Program-O/Program-O.

Petherbridge, Noah. *RiveScript*, July 2009. Accessed September 4, 2016. https://www.rivescript.com/wd/RiveScript.html.

———. "RiveScript." April 2012. Accessed August 21, 2016. http://www.alicebot.org/chatbots32/Chatbots.pdf.

———. *rivescript-python*, 2016. Accessed September 4, 2016. https://github.com/aichaos/rivescript-python.

———. *RiveScript Working Draft*, July 2009. Accessed August 25, 2016. https://www.rivescript.com/wd/RiveScript.

Python Software Foundation. *pip*, March 2016. Accessed September 4, 2016. https://pypi.python.org/pypi/pip.

Quality Health. *2014 National Cancer Patient Experience Survey National Report*. Technical report. Quality Health, September 2014. Accessed August 17, 2016. https://www.quality-health.co.uk/resources/surveys/national-cancer-experience-survey/2014-national-cancer-patient-experience-survey/2014-national-cancer-patient-experience-survey-national-reports.

Reeves, Jack. "Appendix D: The Source Code Is the Design." In *Agile Software Development Principles, Patterns and Practices*, 517–524. 2001.

Rehurek, Radim. *Making sense of word2vec*, December 2014. Accessed August 22, 2016. http://rare-technologies.com/making-sense-of-word2vec/.

Rehurek, Radim, and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora." In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 45–50. May 2010. Accessed August 22, 2016. http://is.muni.cz/publication/884893/en.

Reitz, Kenneth, and Tanya Schlusser. *The Hitchhiker's Guide to Python: Best Practices for Development* [in English]. 1 edition. O'Reilly Media, September 2016. ISBN: 978-1-4919-3317-6.

Riedel, Sebastian, Guillaume Bouchard, Sameer Singh, Matko Bošnjak, and Tim Rocktäschel. *Bloomsbury AI*, 2016. Accessed August 17, 2016. http://bloomsbury.ai/.

Rowe, Jackie. "Introduction to eHNA and Care Planning." *Mac Voice, the magazine for Macmillan professionals* Winter 2014. Accessed August 17, 2016. http://www.macmillan.org.uk/aboutus/healthandsocialcareprofessionals/newsandupdates/macvoice/winter2014/introductiontoehnaandcareplanning.aspx.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence A Modern Approach*. Englewood Cliffs, New Jersey: Prentice-Hall, 1995.

Schlaich, Marc. *pytest-cov*, 2016. Accessed September 4, 2016. https://pypi.python.org/pypi/pytest-cov.

Schreiber, Jacob. *pomegranate*, August 2016. Accessed September 4, 2016. https://github.com/jmschrei/pomegranate.

Sebastiani, Fabrizio. "Machine Learning in Automated Text Categorization." *ACM Comput. Surv.* 34, no. 1 (March 2002): 1–47. ISSN: 0360-0300, accessed August 18, 2016. doi:10.1145/505282.505283. http://doi.acm.org/10.1145/505282.505283.

———. *Text Classification via Supervised Learning: Techniques and Trends*. Lecture, 2011. Accessed August 19, 2016. http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/ir/ir10/slidescorsoferraginashort.pdf.

Shang, Lifeng, Zhengdong Lu, and Hang Li. "Neural Responding Machine for Short-Text Conversation." In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 1577–1586. Association for Computational Linguistics, July 2015. Accessed August 22, 2016. http://arxiv.org/pdf/1503.02364v2.pdf.

Smith, Dustin. *stanford-corenlp-python*, April 2012. Accessed August 20, 2016. https://github.com/dasmith/stanford-corenlp-python.

Sordoni, Alessandro, Michael Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie, Jianfeng Gao, and Bill Dolan. "A Neural Network Approach to Context-Sensitive Generation of Conversational Responses." In *Proceedings of the North American Chapter of the Association for Computational Linguistics – Human Language Technologies 2015 Conference*. Denver, Colorado, USA, June 2015. Accessed August 22, 2016. http://arxiv.org/pdf/1506.06714v1.pdf.

The Economist. "Bots, the next frontier." *The Economist*, April 2016. ISSN: 0013-0613, accessed August 17, 2016. http://www.economist.com/news/business-and-finance/21696477-market-apps-maturing-now-one-text-based-services-or-chatbots-looks-poised.

Tomer, Diwank Singh. *pyAIML*, July 2014. Accessed August 21, 2016. https://github.com/creatorrr/pyAIML/commits/master.

Vinyals, Oriol, and Quoc Le. "A Neural Conversational Model." ArXiv: 1506.05869, *arXiv:1506.05869 [cs]*, July 2015. Accessed August 22, 2016. http://arxiv.org/abs/1506.05869.

Viv Labs. *Viv*, 2016. Accessed August 17, 2016. http://viv.ai/.

Wallace, Richard S. "AIML 2.0 Working Draft." March 2014. Accessed August 21, 2016. https://docs.google.com/document/d/1wNT25hJRyupcG51aO89UcQEiG-HkXRXusukADpFnDs4/pub.

———. *Free A.L.I.C.E. AIML Set*, 2011. Accessed September 4, 2016. https://code.google.com/archive/p/aiml-en-us-foundation-alice/.

Wang, Tong, and Graeme Hirst. "Exploring patterns in dictionary definitions for synonym extraction." *Natural Language Engineering* 18, no. 03 (July 2012): 313–342. ISSN: 1469-8110, accessed August 22, 2016. doi:10.1017/S1351324911000210. http://journals.cambridge.org/article_S1351324911000210.

Watson, Amanda. "Transforming care using eHNA." *Mac Voice, the magazine for Macmillan professionals* Winter 2014. Accessed August 17, 2016. http://www.macmillan.org.uk/aboutus/healthandsocialcareprofessionals/newsandupdates/macvoice/winter2014/transformingcareusingehna.aspx.

Wilcox, Bruce. "Beyond Façade: Pattern Matching for Natural Language Applications," March 2011. Accessed August 21, 2016. http://www.gamasutra.com/view/feature/6305/beyond_fa%C3%A7ade_pattern_matching_.php.

———. *ChatScript*, August 2016. Accessed August 21, 2016. https://github.com/bwilcox-1234/ChatScript.

———. *ChatScript Advanced User's Manual*, August 2016. Accessed August 25, 2016. https://github.com/bwilcox-1234/ChatScript/blob/master/DOCUMENTATION/ChatScript%20Advanced%20User%20Manual.pdf.

———. *ChatScript Basic User's Manual*, July 2016. Accessed August 25, 2016. https://github.com/bwilcox-1234/ChatScript/blob/master/DOCUMENTATION/ChatScript%20Basic%20User%20Manual.pdf.

Yang, Yiming. "An Evaluation of Statistical Approaches to Text Categorization" [in en]. *Information Retrieval* 1, nos. 1-2 (1999): 69–90. ISSN: 1386-4564, 1573-7659, accessed August 18, 2016. doi:10.1023/A:1009982220290. http://link.springer.com/article/10.1023/A:1009982220290.

Yang, Yiming, and Xin Liu. "A Re-examination of Text Categorization Methods." In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 42–49. SIGIR '99. New York, NY, USA: ACM, 1999. ISBN: 978-1-58113-096-6, accessed August 24, 2016. doi:10.1145/312624.312647. http://doi.acm.org/10.1145/312624.312647.

Yuan, Michael. *Chatbots made for healthcare — Tincture*, April 2016. Accessed August 17, 2016. https://tincture.io/chatbots-made-for-healthcare-fec631bc8462.