
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


August 30, 2016

**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 (Mac Voice, 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[1]. With applications of artificial intelligence to using natural language inputs for different purposes, including general purpose mobile device interfaces[2]. 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[3]. 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 ubiqui-

[1]Numerous articles, among which: (The Economist, 2016), (Berger, 2016), (Knowledge@Wharton, 2016), (Finextra Research, 2016)

[2](Viv, 2016), (Dillet, 2016)

[3](Pandorabots, 2016), (Riedel et al, 2016), (Microsoft Cognitive Services, 2016)

tously in human-computer interfaces and everyday use of computing devices today.

This project is part of PEACH: Platform for Enhanced Analytics and Computational Healthcare (Project 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

### 2.1.1 Macmillan Cancer Support

The eHNA system represents the background project against which the PEACH chatbot team efforts have kept constant reference to from the project inception throughout development.

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 (Mac Voice, 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 information extracted is first stored in a Macmillan data store, external to the NHS N3 network (NHS, 2016). At this stage, Macmillan data storage synchs within a minute with data storages inside N3 and deletes all identifying patient information from the data is anonimized and data about the concern is retained by Macmillan to gather insight into the needs of cancer patients (consent is explicitly required from the patient in order to undertake the eHNA and information about the use of the data is transparently provided).

The front end of the system is implemented as web-app, built using HTML and JavaScript. 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. Each category may itself be a subcategory of the following major topics:

- 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 project. Just

before the start of the project, when teams and roles had not yet been defined, the whole team underwent training about handling patient data and the relevant legislation in the UK (the specific certificates obtained by the author can be found in the Appendices [#MAKESURETHISHAPPENS]).

The following is a summary of key policies the author became familiar with before starting the project, with references to how in particular they affected design and implementation decisions.

Generally speaking, authorization must be provided before any information provided by the patient can be used in any way except the specific purpose of their healthcare. This severly limits the possibility of using third party services.

First, there is no guaranteed that the information can be transmitted securely to the external system. Secondly, this increases the risk of loss and inappropriate use of the information, both due to mishandling by the third party (whether intentional or accidental) and by increasing the risk that people unaware of the relevant legislation may come into contact with the data.

### 2.2.1 The Natural Duty of Confidence

Under UK common law, information that *can* reasonably be expected to be held in confidence under the circumstances (such as the information provided by patients to a clinician), *must* be held in confidence. This applies regardless of whether the information is specifically relating to the patient's physical health, and applies to any practical or other concern the patient may express.

Duties are sometimes contrasted with obligations in the sense that an obliga-

tion is a voluntary covenant a person enters, whereas a duty applies to the person regardless. This means that any personnel (including data scientists and software developers) who work with NHS patient data can be liable for misuse of the data even if they are not formally contracted.

### 2.2.2 The Data Protection Act 1998

The DPA (Data Protection Act, 1998) describes eight principles meant to ensure confidential data about (living) individuals is treated with fairness, and applies to any organization handling such data (e.g. financial institutions).

The nature of information covered by the act is "sensitive" in the sense that it may be used in ways that affect the subject to significant extents. Identifying information (such as name and date of birth) is normally regarded as such.

The second principle of the DPA specifies that the purposes for which personal data is being gathered have to be transparently described to the person. This means that information provided by a patient for the purpose of their own health care can only be used for this purpose and no other (including mass aggregation of data to gather insight for any purpose from third parties involved).

The eight principle also requires that personal information is not sent outside the European Economic Area in most cases, which can also cause problems with the geographic location or accessibility across the world of data shared with third parties.

### 2.2.3 Conclusion

Patient consent should be gathered explicitly, having clearly explained all of the purposes for which the data may be used, before any information about them can be processed (with few exceptions, for example where the information becomes critical to national security and similar cases).

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, and provided a special agreement (such as a Data Transfer Agreement) has been brokered to ensure both parties understand the legal and ethical implications of sharing even anonimized data; furthermore, it would be best to also gather consent explicitly from the patient even where the data has been anonimized. In such cases, the duty of confidence 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 externally although the emphasis on clean architecture will allow for such a choice to be made in a future iteration of the chatbot project, where an agreement has been brokered, or authorization is otherwise provided to make use of third party services.

## 2.3 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. Furthermore, to investigate the application of machine learning to extract information from user input that would be relevant to the chatbot system as a whole.

To be specific, it is not so much tasks of information extraction or named entity recognition that were identified as most useful in this case, but rather the possibility to classify user input according to sentiment analysis and text classification. In order to inform the chatbot system reply to the user by providing additional information to the raw user input. Why text classification tasks? Because they may be used 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" (or rather input-pattern driven) approach are used as part of a more complex system.

In the abstract, this is an attempt at exploiting the power for generalization that is characteristic of approaches to artificial intelligence resembling the nature of experience as a way humans acquire knowledge, human capacity (and propensity) for inductive reasoning (Russell and Norvig, 1995, p.592; Biermann, 1986, pp.134-135; Hawthorne, 2014; Hume, 1777, section 4, part 2), while at the same time retaining the rigor and control provided by more rigid rule-based approaches to AI, which would ensure the conversation remains on the range of topics and allows the user to carry out the information gathering phase for the creation of their care plan, where a set of heuristics or rules of

inference is used, these in turn resemble deductive reasoning, where certainty of the conclusion of the reasoning is guaranteed by the soundness of the deductive calculus used, and the mathematical certainty in the premises needed (Russell and Norvig, 1995, pp.163-165, Beall and Restall, 2014).

This approach is in particular to be contrasted with neural networks and other approaches to AI which behave as "black boxes", and cannot by their nature proivde an intelligible explanation of their categorization process (or more generally decision process) as the result of their learning is stored in the form of a weighted graph (Russell and Norvig, 1995, p.567).

### 2.3.1 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). More formally:

For a document $d$ and a set of categories $C = \{c_1, c_2, c_3, ..., c_n\}$, produce a predicted class $c \in C$

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):

$C_{cc} = \{phyisical, practical, family, emotional, spiritual\}$

To be precise, we will be focusing on document-pivoted classification, where we wish to approximate an ideal mapping from the set of documents $D$ to

our $C_{cc}$:

$f : D \mapsto C_{cc}$

The task is normally turned into a supervised machine learning task, by training a model over a set of document-category pairs (Sebastiani, 2011, slide 7, 13):

$\{..., \langle d_i, c_j \rangle, ...\}$

Supervised learning is a form of machine learning where the machine is trained over a set of "hand" labelled examples. The "supervision" consists in already knowing the right answer for each training input, and wanting to use the system to automatically label future instances as desired (Russell and Norvig, 1995, p.528). This is in contrast with other forms of machine learning, such as reinforcement or unsupervised learning, where the answer is either unkown or "fuzzy" unlike with supervised learning. Take for example a robot navigating an industrial warehouse, as the surrounding circumstances change, the behaviour desired cannot simply be evaluated in binary terms (i.e. either "good" or "bad") because the problem of navigating a busy environment is by its nature not binary, there is in most cases a continuous spectrum of evaluation.

A model is trained over the training set and then tested against an unseen test set, also made up of hand-labelled samples. The model classifies each test sample and evaluation metrics can be drawn from comparing the model classification with the known ("gold") standard for the sample.

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. Different features will be relevant to different document classificaiton tasks, for example certain words may occurr more frequently in positive movie reviews as opposed to negative movie reviews, but those particular words are unlikely to also be indicative of whether the person who wrote the document happened to be male or female.

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). There are several categorization algorithms that may be employed, discussion is deferred to Chapter 4.

### 2.3.2 Sentiment Analysis

The task of sentiment analysis was also considered important for the chatbot. The insight to extract from the raw natural language user input was deemed important to the assessment of the user distress level with respect to issues the user would be discussing with the chatbot.

As described in Chapter 3, however, this task became secondary to the purpose of the first implementation, and was therefore never implemented. Nevertheless, for future references, the theoretical resources discovered as part of the background investigation for the current project.

Sentiment analysis can be seen as a type of text classification where the document is to be classified into categories of either positive or negative feeling ($C_{sa} = \{positive, negative\}$).

Therefore, a simple sentiment analyzer can be nothing more than a text categorizer, which may be built through the same set-of-words approach

outlined at the end of the previous section (this approach is sometimes called "sentiment classification", Liu, 2010, p.628, 637–). This simple approach has obvious shortcomings: for one, some words change polarity (postive or negative) depending on the context (Ding, Liu and Yu, 2008, abstract). For example, the word "great" may in general be more likely to be found in positive as opposed to negative movie reviews (for example), but it is easy to imagine a negative review where the word also occur (e.g. "great disappointment").

Secondly, the sort of documents we are interested in analyzing will be for the most part short sentences, part of a conversation. Hence the additional difficulty of having to keep track of the present topic given the conversational context: the "volleys" of messages exchanged between user and system, as well as which user inputs are relevant to one topic rather than another.

A more interesting approach may involve, instead, first deciding whether the document is expressing a subjective evaluation or an objective statement ("I hate the world" contrasted with "it is raining"), and if the first, then whether the polarity of the statement is either positive or negative (Liu, 2010, pp.640–). This approach may be relatively easy to implement (compared to more advanced information retrieval approaches, that may want to also extract what the object of the sentiment is, and what the nature of the sentiment is) and may enable some of the distresses of the patient and the object of these to be extracted from the conversation. As we will see in the Requirements section of the report (Chapter 3), the requirements for the current project are significantly more modest than that.

### 2.3.3 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 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, and contribute to the creation of shared software packages for the PEACH project.

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, Klein and Loper, 2015). The decision was also made on the basis of the expected needs of the individual subprojects based on the requirements gathered in collaboration with Dr Ramachandran: the NLTK is a versitile, general purpose suite of NLP tools. 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 for most of its NLP needs. No other frameworks were used for NLP general purposes specifically

(such as tokenization and stemming, even classification), althought other tools where used in areas related to NLP, for the implementation of the chatbot brain and for the generation of synonyms, as reported below.

### 2.3.4 Categorizing text with the NLTK

The NLTK documentation provides a well written introductory tutorial to text categorization and supervised learning (Bird et al, 2015, Chapter 6). The author found helpful in particular the way the train/dev/test split is justified and used (Ibid, section 1.2).

In supervised machine learning generally, the set of all data points available is normally split into three disjunct subsets. The training set is used to create a representation, internal to the classifier, of the propeties of the data points that are relevant to deciding their categories (the "features"). This is normally the largest set in the split. The development set is used to improve the feature extraction: the example in Chapter 6 of the NLTK book shows how the performance of a classifier can be improved in this way (see also Manning et al, 2009, pp.271–).

The task is classifying proper names into genders. At first, the only feature that is considered is what the last letter of the name is, with the classifier surveying the distribution of letters in the training sample and "learning" what the proper label is by being given the answer. Then, feature extraction is improved by looking also at other features, such as the length of the name and what the initial letter is.

Thus the development set is used in an intermediate testing stage, before

proceeding to test the classifier against the yet unseen test subset of the original set of all data points.

The NLTK exposes a range of natural language corpora and a machine learning package (aptly named "classify") where implementations of single and multi-category classifiers can be found (https://github.com/nltk/nltk/tree/develop/nltk/classify). A "corpus" is a collection of related natural language resources, used for a variety of purposes in NLP, such as training of classifiers (SIGLEX, undated). For example, it is possible to train a sentiment analysis classifier over a corpus of movie reviews.

Of interest to the present projects where the ready available implementations of various types of single-label classifiers, and the intuitive API they expose. These classifiers are implemented as simple Python objects, to be instantiatiated, trained and producing a label for an input after training.

These classifiers are very easy to use when data is available. An obvious problem with the present project is that no data of the relevant shape was available whatsoever. 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 (a member of $C_{cc} = \{phyisical, practical, family, emotional, spiritual\}$, formally).

Therefore, the author created a survey that Dr Ramachandran was kind enough to mandate completion of through the team (the survey can be found here: https://goo.gl/forms/ylkI50XckV9yvCCm2). This resulted in

250 data points. By data point here is meant a single unit of data relevant for the purpose, in this case, a chat message - label pair. Given the time, however, that it took for the collection of these data points, the project was out of time before they could be put to use with the current implementation, unfortunately.

Even if a classifier was trained and saved over a suitable split of this data, it would still be necessary to go through a development stage, where the feature extractor was further and further refined, to improve the performance of the classifier against the dev set, before moving on to evaluation proper.

Conveniently, feature extractors can be used with an NLTK classifier instances as simple functions (or lambda expressions) that return a dictionary (or map) of arbitrarily named features to values. More specifically, an NLTK classifier takes its input (or set of inputs) as instances of such maps.

### Advanced Research into Categorization

So far, we have looked at how classifiers would be used to give labels to distinct documents, however, as noted earlier, 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 many, 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.

### 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, 2015, 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 $i$: $b_i : O \mapsto [0, 1]$

The tasks that Jurafsky and Martin identify for such models are three (ibid,

p.6):

- Likelihood: given a model M and a sequence of observations $O$, determine $P(O|M)$
- Decoding: given a model M and a sequence of observations $O$, determine the "best" corresponding sequence of states $Q$
- Training: given an observation sequence $O$ and a set (not a sequence) of states $Q$, train a model M

For our problem: 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 may be.

### The limits of the NLTK

Unfortunately the implementation produced does not make use of sequence classification, but instead uses simple classifiers that consider each user input in isolation. 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 that the focus of the current project is laying down the fundamental architecture of a chatbot for eHealth applicaitons, 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 the implementation 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: https://github.com/nltk/nltk/blob/991f2cd1e31f7c1ad144589ab4d2c76bee05aa7b/nltk/classify/api.py (the link should reference the latest commit to the NLTK project as of 21/08/16).

There are 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 exam-

ple, “pomegranate” here: https://github.com/jmschrei/pomegranate#hidden-markov-models (Schreiber et al, 2016).

## 2.4 The Chatbot

The very high level general architecture of a chatbot system is normally described in terms of (Bird et al, 2015, Chapter 1 section 5.5; Jurafsky and Martin, 2009, pp.857-867):

1. A natural language understanding layer
2. A diaglogue 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). 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).

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. 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.

### 2.4.1 The Chatbot "Brain" Market

This section aims at describing the options that were discovered while researching ways to set up an initial chatbot. A lot of the open source projects available make use of the patter/grammar and template model described above, but there is a notable alternative in the emerging (yet immature) technology of using neural networks or similar tools to build conversational models. Finally, as a last resort, there are AIaaS providers: providers of

remote intelligent agents and related services [1], 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 take care of making 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 standard 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 template. The language was also 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

[1] (Pandorabots, 2016), (Riedel et al, 2016), (Microsoft Cognitive Services, 2016)

input for processing or to repeat back by adding it to 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". There are a number of interpreters available for AiML in various programming languages, and there are freely available "libraries" of AiML files for others to include into their own chatbot (Wallace, 2011; Pandorabots, 2016; 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 the author 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 (at the time of writing) there are no easily traceable AiML interpreters implemented in Python. 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 also supports external procedure calls, wildcards, optional sub-patterns and the other pattern matching features of the previous standards. Something that put the only ChatScript interpreter aside from the other standards examined is that for one, there is only one and no other open source interpreter projects. Secondly, the interpreter is implemented in C++.

**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 (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/changelo 08-15-version-640-current-cjihrig).

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 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 simpler but expressive syntax being the final deciding factor for the current implementation.

## 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.

RiveScript arrays, in particular, could be used to define sets of synonyms, to be reused across multiple matchers.

### 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 ocument 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).

The results were, unfortunately, unsatisfactory, due to the fact that the model when queried for synonyms for a word would often output irrelevant data. 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 was trained with (as a training parameter, see Ellenberg, 2015).

### 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.

### 2.5.4 Dynamically Streaming Data into a RiveScript Interpreter

The RiveScript framework offer ways to stream RiveScript code into a running interpreter (see: http://rivescript.readthedocs.io/en/latest/rivescript.html#rivescript.rivescript.RiveSc
While the synonym generation may be performed offline with files being generated to be afterwards fed to the RiveScript interpreter, it may be

possible and interesting to stream code into the interpreter dynamically. This may allow the system to understand terms it has never seen before and do not currently feature within arrays representing synonyms within the chatbot brain files. A synonimity relationship between the new term and existing chatbot brain data may be discovered and the chatbot brain may be made to dynamically learn new terms, as these are streamed into the running interpreter and also written to secondary storage for use at the next system reboot.

Again, due to the numerous other concerns and the relatively low priority for the synonym generation component, the implementation that currently exists is the one that did not perform satisfactorily, and was not used as part of the full system. The research reported here in this and other topics is for the use of future project iterations.

# 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.

Note that, in accordance with agile methodology principles (as per the author's aims), "architecture" here does not refer *exclusively* to the production of documents such as class and component diagrams, or other UML arti-

facts (Object Management Group, 2015); but rather the *primary* document ideintified as the technical design document produced by the engineering activity is the source code. Just like in other engineering industries, software engineers design a meticulously crafted document that is then passed on to the manufacturing staff, requiring no further input from the designers to produce a concrete artifact. The manufacturing "staff", in this case, being compilers, linkers, interpreters and virtual machines (Reeves, 2001).

## 3.2 Requirements Gathering

The project was first proposed by Dr Ramachandran, who also organized a meeting between 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, the idea of a specialized search engine emerged, with Rim Ahsaini interested in taking charge of that system. 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.

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.

## 3.3 Requirements Listing

}

## 3.4 Use Case Diagram

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 to reply with a generative or non-generative reply | Must |
| RQ3 | The chatbot will decide whether the generative reply should be provided as the result of a query or bot-brain generated reply | Should |
| RQ4 | The chatbot will model the data gathered appropriately for durable storage in datastore | Must |
| RQ5 | The chatbot will interface with the search/query engine component of the larger system it is part of | Should |
| RQ6 | The chatbot may performsentiment analysison user input | Could |
| RQ7 | The chatbot may use the preprocessed input (categorization/sentiment analysis) to inform the bot-brain | Should |
| RQ8 | The chatbot may interface with server side request handling logic | 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 |
| 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?"* The answer that came to me was, *"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 is provided before moving on to the

As mentioned before, the complete application is the product of a team effort, with Deborah Wacks working on the web server implementation and the UX design, Rim Ahsaini working on the specialized search engine, and the

author working on the chatbot brain. The scope of the author work is limited to the implementation of the chatbot and supporting systems. The overall architecture of the components is illustrated below.

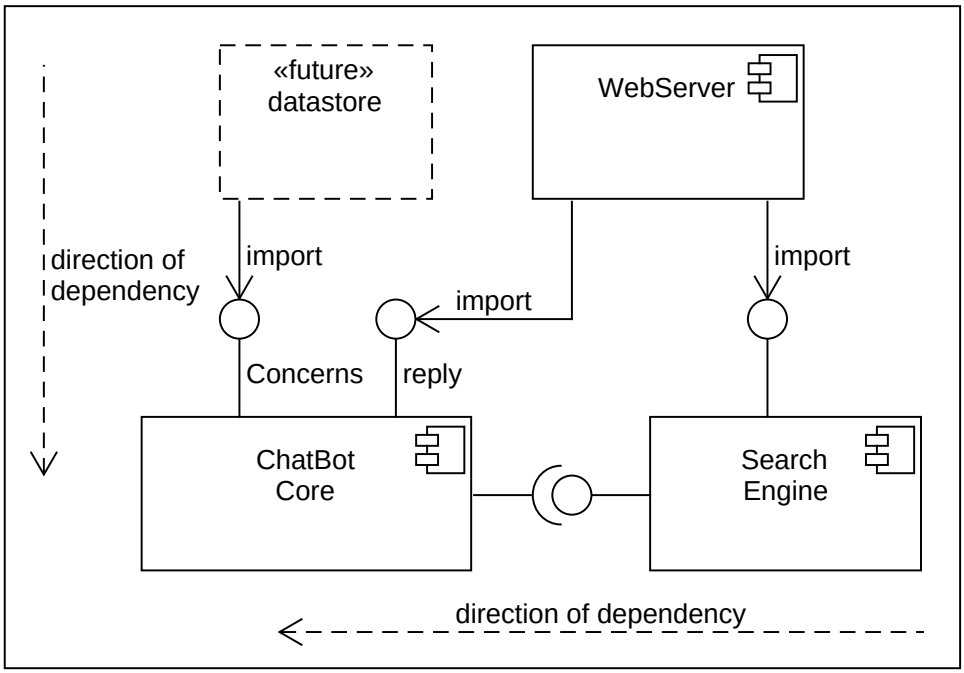


Figure 4.1: System component diagram

## 4.1 Software Architecture

### 4.1.1 Principles of Software Design

SOLID principles of software design were used by the author while working on the source code; these are dependency management principles (policing the "import" statements throughout the project) and clean design principles,

chief among them the Separation of Concerns principle (Martin, 2003, Section 2). Additionally, the code design guidelines advocated by Martin (2009) were discovered by the author during the writing, and an attempt was made to apply them (in line with author's stated aims). 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. Additionally, 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[1].

The core of a chatbot system are the high level abstractions precisely defined in the interfaces and the abstract classes (Dependency Inversion Principle). The boundaries between systems insist that change in the external systems do not affect the well functioning of the core of the system: this should not have dependencies flowing outwards, with outward facing boundaries that expose interfaces for the external systems to implement or use.

### 4.1.2 High Level Structure

The folder structure of the chatbot is meant to be self-describing, the reader should be able to understand what the purpose of the system and of the sub-folders within without further assistance (see Martin, 2011).

Within the botinterface subdirectory we can find the high level abstractions of the core system: the bot_abstract module defines the interface to the bot

[1]These approaches is not free of controversy, but proceed from a very influential and successful school of thought, hence the author is adopting it in an effort to experiment and improve as a software engineer.

application as a whole, 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[2].

The brain folder contains the RiveScript language files that define the matchable patterns and reply templates, the Python macros callable from within the brain, and the topics structure as seen from the brain, that limits the scope of pattern matchers depending on the the stage of the conversation between user and system.

The messagelog package defines a data model that may be useful to gather the data about a conversation a user had with the system and extract the information to persist for the purpose of the care plan creation. Again, the scope of the project is limited to the architecture of the system, and while the larger PEACH team produced a relational database schema to persist the data, it is not within the objectives of the author's project to follow through with serializing the information to durable storage. The package is there to be used in the future.

The concerns package encodes the information about a user's concerns, as well as the range of acceptable macro and micro topics (or categories) for each concern. The point of access to this subsystem is defined in the *drive_conversation_abstract* module, as another interface. This interface describes the API to the part of the module that keeps track of what the state user concerns are, and their priority order. The concerns_factory module defines a class with static (or class-level) methods for the purpose of keeping track of various user sessions in the system, each represented by a *DistressCon-*

[2]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.

*versationDriver* object (a concrete implementation of the ConversationDriver interface).

The categorize package exposes a set of modules to train, store to the file system and retrieve evaluation metrics for single label classifiers. This package is the least well designed element of the system primarily because it was the first to be developed. The reason this subpackage was not used in the final pipeline implementation has to do with the fact that it took too long for the survey that was created to collect relevant data to produce results (#REFERENCERELEVANTPART). 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.

Finally, the synonym package defines a SynonymExtractor interface and one implementation for it. As noted in Chapter 2, this implementation did not perform to expectations, and was therefore not used in the current system. However, the synonym package should be easy to extent to allow (for example) a WordNet synonym generator implementation, and the package already provides logic to represent the RiveScript syntax data structure (the RiveScript "array") and a way to write this structure to file[3].

[3]Additionally, see "Dynamically Streaming Data into a RiveScript Interpreter" in this report for instructions on how to stream RiveScript syntax into a running interpreter.

### 4.1.3 The BotInterface, MessagePreprocessor and MessagePostprocessor layers

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

This provides a degree of decoupling from the chatbot framework, 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 (in this sense it is a PROXY: it "looks" like the underlying framework). 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.

```
def reply(self, message):
      userid = message.getUserid()
      messagecontent = self._preprocess(message.getContent())
      reply = self._interpreter.reply(userid, messagecontent)
      reply = self._postprocess(reply)
```

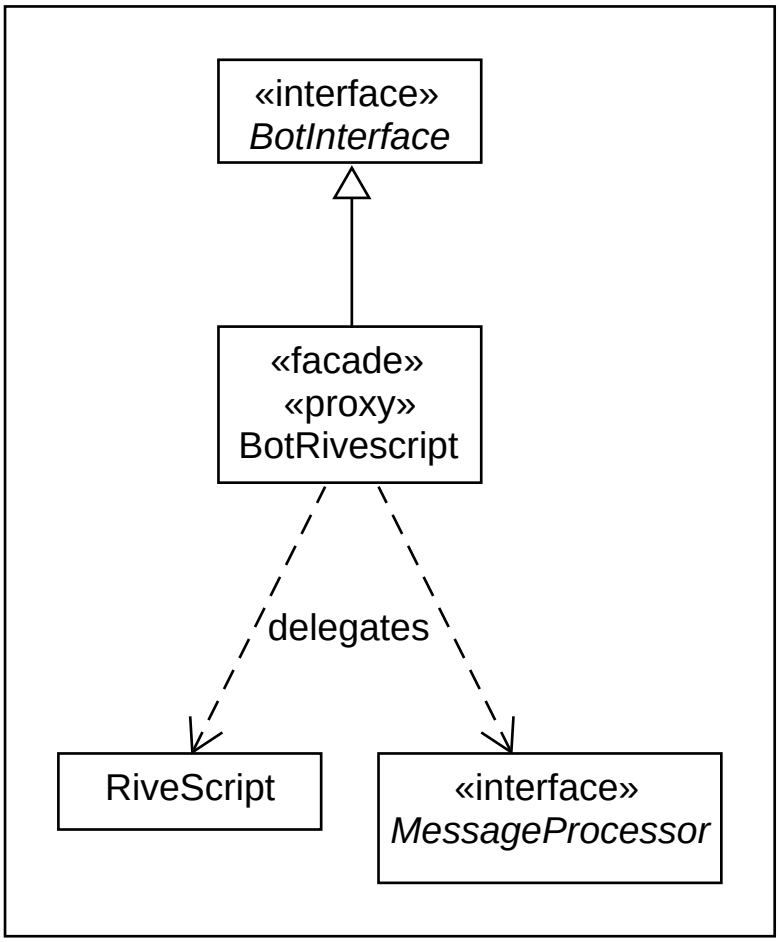


Figure 4.2: BotRivescript class

```
    return reply
```

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. The Dependency Inversion principle states that the direction of dependency in the core application logic goes towards the abstract and not the concrete. However, if all our dependencies point at the abstract (which cannot be instantiated) then how are we going to instantiate them? Hence the need for creational logic that can limit the number of concrete subclasses of the abstract type the caller needs to know about.

The rivescript_loader module is responsible for loading the chatbot brain files from the host operating system, and produce errors as needed. This makes it easier to mock the chatbot framework association of the BotRivescript class.

The façade delegates to a message preprocessor to process the input to the system and a postprocessor to process the system output. Looking at the corresponding classes, we notice both of these inherit from the MessageProcessor class, which is another interface. Looking at the the preprocessor we can see that it 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. This helps with RiveScript, as certain input patterns that should be matched do not get matched if the user mispells a word, uses a declination of a verb or adds stopwords (such as determinants) which had not been anticipated in the grammar.

For example, in RiveScript, the input pattern in the following rule:

```
+ I like you
- I like you too
```

would not match the input “I am really liking you”, even though the programmer may have intended for general expressions of affection to trigger the response. Stemming reduces verbs to their root: “liking” or other declinations are reduced to “lik”. Stopwords removal will eliminate words that may cause matching to fail, even though they do not modify the meaning of the sentence significantly, for example “I do like you” (auxilliary verbs are used in English for emphasis).

Modules responsible for the preprocessing are defined within the “preprocess” subpackage. The creation of stemmers is handled through a factory that is minimal for the current implementation, but could in the future include logic to select stemmers on the basis of a simple parameter input, this would help decouple the caller from the concrete implementation of the stemming interface (as above). Notice how the dependency to the nltk package is limited to only the concrete implementation that makes use of it in this class hierarchy. Just like with the chatbot framework, 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 “homebrewed” solutions may be used instead.

More interesting is the application of the TEMPLATE METHOD pattern in the StopwordRemover abstract class (Gamma et al, 1995, pp.325-330). We can see here how the stopword remover implemented in the nltk was too strict and would remove semantically meaningful tokens such as “no”, “not” and

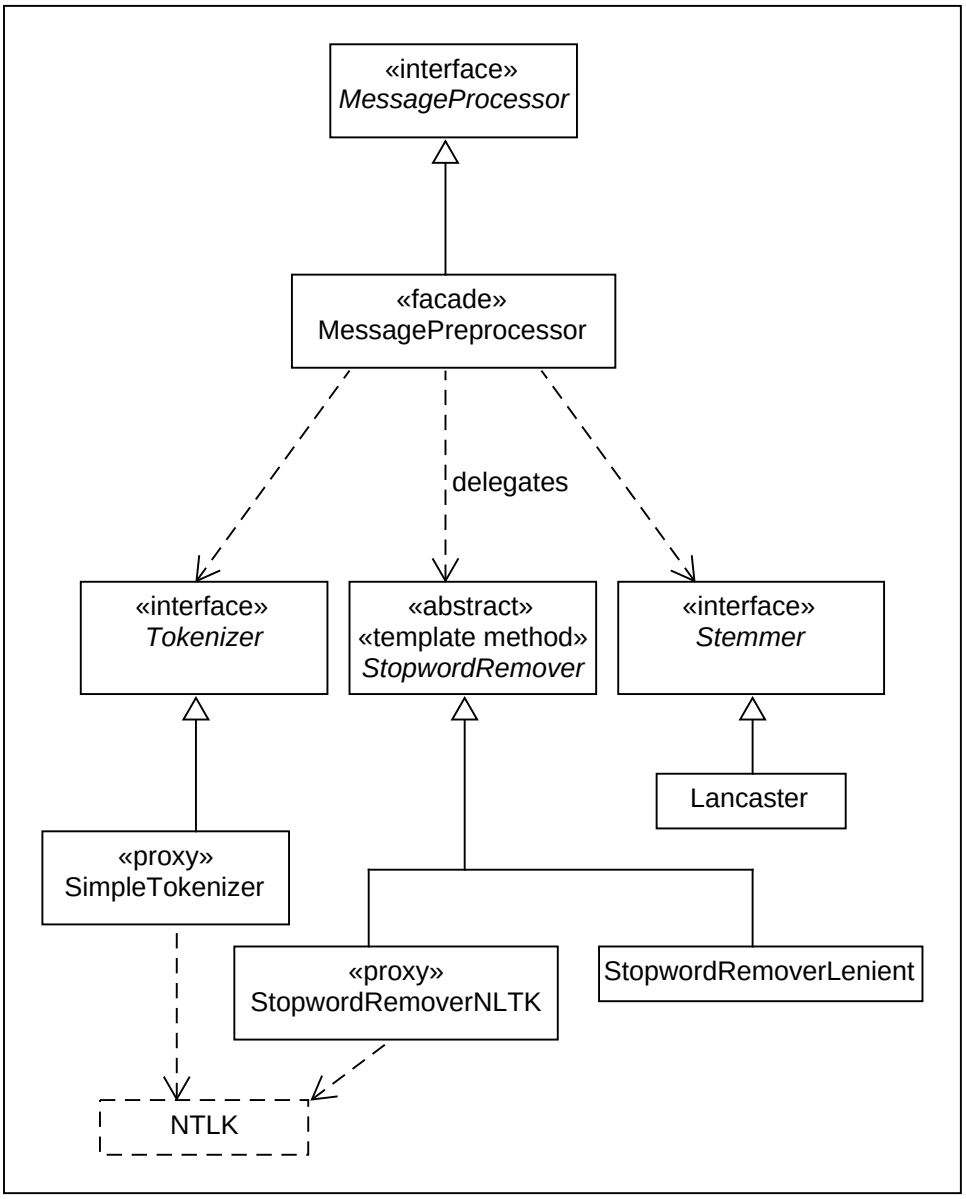


Figure 4.3: Preprocessor class diagram

declinations of “to be” (presumably on account of the fact that they are also used as auxilliary verbs and would create noise in information retrieval tasks).

Therefore, using this stemmer, the input “I do not like you” would have matched the rule:

```
+ I like you
- I like you too
```

which may not have been the programmer’s intent. Therefore, the stopword removal algorithm expressed in the StopwordRemover abstract class relies on subclasses exposing an iterable describing the stopwords to remove from the tokens.

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 search engine being worked on by Rim Ahsaini.

The modules it defers to apparently 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, with 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

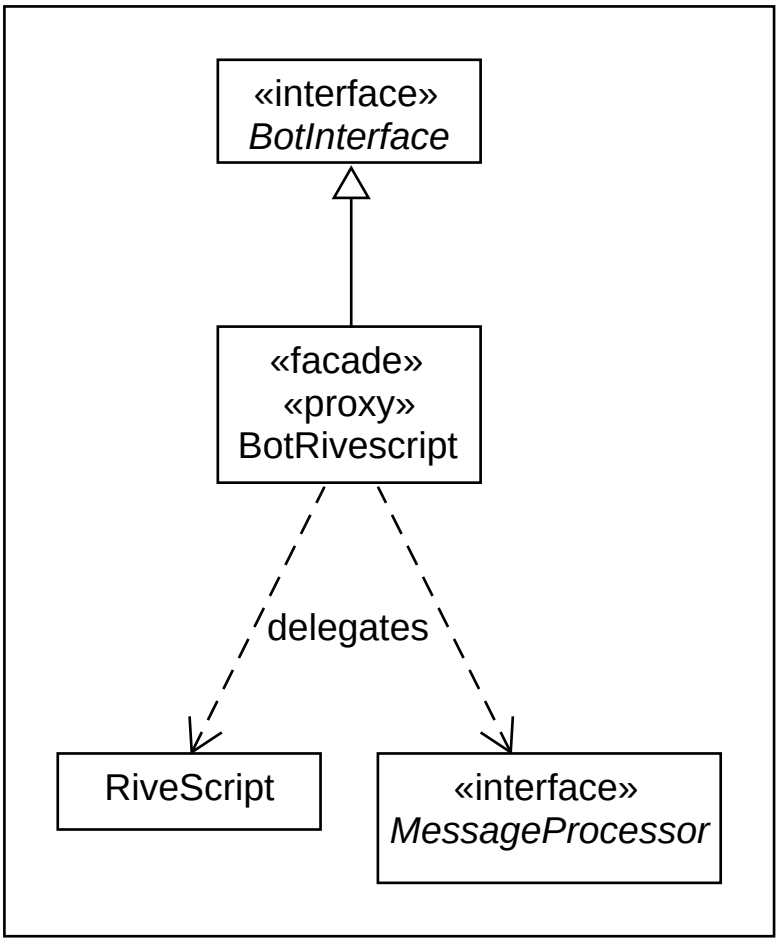


Figure 4.4: Postprocessor class diagram

an ADAPTER pattern sitting at the boundary between the systems enforces compliance with the 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).

This concludes the explanation of the main message system exchange pipeline. This model allows us to keep the core framework "at arms length", as opposed to tightly coupling with it and making it almost impossible to replace it in the future, and in fact it afforded us an intuitive way to overcome the limitations of the framework and to integrate with an external system. Finally, it exposes a minimal but complete API to the caller.

### 4.1.4 The Brain Implementation

The brain is implemented as a set of RiveScript files, split by categories with the exception of the begin.rive, the global.rive, and the python.rive files.

These files play each a special role: begin.rive defines the core grammar substitutions and basic synonyms as arrays, independent of the topic. The 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.

We will look at the positives and negatives of the brain implementation in order.

### The Good

RiveScript provides very useful tools in the ability to call Python (and other programming languages) from inside reply templates. This enables the programmer to draw information from outside the interpreter and also perform operations on behalf of it. This was instrumental in using the external data model provided by the ConversationDriver data model to drive the conversation forward. Secondly, RiveScript can store state that is only accessible from within a specific user session, and manages user sessions internally by requiring that both the user input and a userId (created by the caller) be passed when asking the RiveScript for a reply. This is also very useful to store information about the current topic in the RiveScript. Finally, the topic inheritance model allows us to define a global scope, wherein we define the matchers that are used to perform topic-invariant operations (such as changing the topic or moving on to the next concern).

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 Bad

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 of the present project: the system architecture.

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 RiveScript class, and issues with inconsistent internal state within the framework internal 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.

### 4.1.5 Data Models for Concerns and Messages

The concerns and messagelog packages define data models to respectively represent the user concerns so that these can be accessed from the chatbot brain and used to decide how to move the conversation forward, and provide a log of conversations between user and system so that these can be retrieved and stored at a later time.

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, ConversationDrivers) for all users.

### 4.1.6 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 is primarily its capacity to easily split and categorize several sentences, something which could be achieved with the NLTK, but not with as much abstraction as provided by TextBlob (https://textblob.readthedocs.io/en/dev/classifiers.html#classifying-

textblobs).

The package provides a convenient 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 abstraction in the form of an interface or a façade. This was the product of the author learning Python, becoming more comfortable with the applicaiton of software design principles and working on the package at the same time.

### 4.1.7 Synonym Generation and Automated RiveScript Generation

The final module examined is the synonym generator. The subpackage (as others here described) exposes an interface describing the high level abstraction of the subsystem.

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

As it is possible to see from the integration tests (tests/tests_integration/test_synonym/) the cosin distance does not always appear to capture semantic synonymity (Mikolov, 2015; McCormick, 2016b; ). It is possible to see, for example, that 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.

As mentioned in the discussion, the implementation of this concrete class was done with the help of the Gensim library (Rehurek, 2014; Rehurek and Sojka, 2010; McCormick, 2016a). The reader is redirected to the discussion of alternatives in Chapter 2.

As far as generating RiveScript, the syntax of the language is really simple, and it is not difficult to generate syntactical arrays. The rivescript_writer module already implements this behaviour. 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.

## 4.2 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 implementations provided.

# 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 brain matchers, redirects and coditionals. This chapter also describes the process of classifier evaluation provided by the classification package.

## 5.1 Test Driven Development

TDD is described as a discipline prescribing adherence to three rules whenever writing code (Martin, 2009, pp.122-123):

1. Do not write any production code before you have a failing unit test

for that code.

2. Do not write more of a test than is sufficient for it to fail (not compiling counts as failing).
3. Do not write more production code than is sufficient to pass the test.

There are three main benefits associated with this practice. The first is that it encourages to take on the perspective of the caller of the code, before any code has been written. This makes the code easy to call and use to the programmer who is going to consume it.

Secondly, it encourages a loosely coupled design. This is because a *unit* test is meant to test a unit of code (be it a procedure, a subroutine, function or class) in isolation; it is the most focused type of test possible. This means that any external dependencies, interactions and collaborations ought to be mocked for the purpose of the unit test. 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 any external interference.

Third, the source code always tells the truth. Failing all other methods of documentation, from diagrams to Python Docstrings or Javadoc, the source code is the ultimate documentation of a software project. Unit and integration tests provide the documentation that describes how to use the system and any and all failure modes or edge cases for that system. Furthermore, 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.

For example, the _sendUserMessageAndLog() function of the messagelog package integration test demonstrates the way to use both the chatbot and

message logging facilities provided by the present package in a way that is technically accurate to the point that it could be compiled to production, and intuitive enough that programmers looking at it should be able to understand how to use those facilities without further guidance[1].

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

## 5.2 The Project Tests

The truth about the 120 unit tests in the present project is that the author was learning what TDD was during development, hence not all tests were written according to the three "rules" of TDD. A test-first approach to software development, however, has been taken from the start (Beck and Andres, 2004, pp.50-51, 97-102). As a result, the later a unit was designed, the better the tests for it were.

Integration tests were produced to test the interaction of different units. These were not always pairwise (two units at a time), but were carried out depending on the interactions the author was concerned with documenting and verifying (McConnell, 2004, p.499). The 66 integration tests include numerous tests of the chatbot brain, piped throught the complete BotRivescript façade.

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

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.

One final note about the tests: the pre-trained word2vec model used in the integration tests which is necessary for the Word2VecSynonymExtractor implementation to work (not provided with the system, but obtainable from: https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit) is very demanding in terms of main memory, likely to cause thrashing on even powerful machines, see appendix A for more details.

(#ADD PYTEST COVERAGE REPORTS)

## 5.3 Categorization Evaluation

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 against the expected label (Bird et al, 2015, Section 3.2). It may also be useful to build a "confusion matrix" (ibid, Section 3.4).

In this matrix, the diagonal represents the percentage of correctly labelled samples, the column for each category for cells not lying on the diagonal represent the percentage of times the labelled indicated by the column was mistaken for the labelled indicated in the row. A simple graphical representation of such a matrix can be obtained via the *confusionmatrix* module of the *metrics* subpackage of the NLTK:

http://www.nltk.org/_modules/nltk/metrics/confusionmatrix.html.

The way the categorization package provides an evaluation of the classifier is through the ClassifierEvaluator class. This class' interface 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 multi-lable 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).

Because of the fact that all of the data extracted via the questionnaire (see Chapter 2) has a balanced number of expected labels for categories, we are not concerned about the training data being skewed with low positive cases for any one category, micro-averaging was chosen as the balance of effectivness for all categories, with the F measure being the final combined evaluation measure.

(#PENDING IMPLEMENTATION OF MICROMACROAVG)

# Chapter 6

# Conclusion

This chapter reviews the project goals and aims, finally, the author expresses his thoughts on the project going forward, informed by a critical review of the open source tools used.

## 6.1 Project Goals Review (#MAP TO REQUIREMENTS)

- **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.

- **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 "Mock-Search" class).

- **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. Secondly, NLP was used for the synonym generation.

- **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.

- **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.

## 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 a Java Swing interface. 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. 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.

### 6.3.1 Evaluation of Technologies Used

#### RiveScript

While the choice of RiveScript as the chatbot brain framework was sensible at the time given the prior research, the author reccomends other options are investigated during the next iteration, for the following reasons.

The undocumented problems encountered with it during development, its limitations with respect to matching semantic patterns of input (unlike others like ChatScript), the chatbot brain implementation provided with the current project is not very extensive.

### NLTK and Gensim

The NTLK proved to be a versatile tool, used primarily in the categorization and message processing modules. Its main limitation is the lack of sequence classifier support (as lamented in Chapter 2). For this reason it is recommended that alternatives that offer such models are investigated during the next iteration (see for example Schreiber et al, 2016).

The use made of Gensim in the current iteration was very minimal, but this tool seems so highly specialized that it is difficult to find alternatives to it.

# Appendix A

# System Manual

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

## A.1 Installation

The full project source code (including the code of the other team members) is available at: https://github.com/andreallorerung/peach-chatbot-alpha. The author's contribution to the project can be found under the '/FlaskWebProject/chatbot/' directory therein (https://github.com/andreallorerung/peach-chatbot-alpha/tree/master/FlaskWebProject/chatbot). This can also be found at https://github.com/nterreri/peach-bot.

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.

When wanting to download the submodule after having cloned into the repository non-recursively, use *$ git submodule update –init –recursive* to download the submodule.

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 memebers 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[1]. The most straightforward way to install the dependencies is to 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/
```

[1]https://en.wikipedia.org/wiki/Pip_(package_manager) Python Software Foundation, 2016

### A.2.1 The *botinterface* package

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 *botinterface/bot_builder.py* package 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. Should that not work, 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 chatbot component.

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.

### 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 (on modern operating systems) 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 mentioned in Chapter 5, this particular implementation was underperforming, but in the future may provide the blueprint to other better performing implementations.

## A.3 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 as follows, 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. It takes around 3-4 minutes on an 8-core desktop with 8GBs and the amount of memory required to run it *without* thrashing happening as pages are swapped in and out of virtual memory is higher than 8GBs.

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 mere seconds (as it should).

The unit tests, in fact, are designed for quick execution (they are meant to be ran many times an hour during TDD; see F.I.R.S.T. Martin, 2009, p.132). Running these tests should take less than a minute. Finally, if the pytest-cov package has 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. See (#WHICH-LETTER) below for results collected.

The reader is redirected to Chapter 5 and Appendix (#WHICHLETTER) for more information about the tests.

# Appendix B

# Tests Results

Unit tests coverage for the *botinterface* package:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                    Stmts   Miss  Cover
-------------------------------------------------------------
botinterface/__init__.py                    0      0   100%
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%
-------------------------------------------------------------
TOTAL                                     139     12    91%
```

Unit tests coverage for the *categorize* package[1]:

```
---------- coverage: platform linux2, python 2.7.9-final-0 -----------
Name                                      Stmts   Miss  Cover
-------------------------------------------------------------
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%
-------------------------------------------------------------
TOTAL                                        91     16    82%
```

Unit tests coverage for the *concerns* package:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                      Stmts   Miss  Cover
-------------------------------------------------------------------
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%
```

[1]You may notice the different platform running the tests for the categorizer (linux) this is due to an issue with the pickle package on Windows, see: https://github.com/DataTeaser/textteaser/issues/3

```
concerns/rivescriptmacros.py                  24      3    88%
concerns/topics.py                            14      0   100%
-------------------------------------------------------------
TOTAL                                        158      9    94%
```

Unit tests coverage for the *messagelog* package:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                  Stmts   Miss  Cover
-------------------------------------------------------
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%
-------------------------------------------------------
TOTAL                                    35      0   100%
```

Unit tests coverage for the *postprocess* package:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                     Stmts   Miss  Cover
----------------------------------------------------------
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%
```

```
-------------------------------------------------------------
TOTAL                                      79      6    92%
```

Unit tests coverage for the *preprocess* package:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                     Stmts   Miss  Cover
-------------------------------------------------------------
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%
-------------------------------------------------------------
TOTAL                                       61      7    89%
```

Unit tests coverage for the *synonym* package:

```
---------- coverage: platform cygwin, python 2.7.10-final-0 ----------
Name                                   Stmts   Miss  Cover
-------------------------------------------------------------
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                                       55     18     67%
```

Total:

- 527 executable statements
- 52 missed statements
- 88.83% averaged percentage

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

```
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.

The tests also test the RiveScript brain data, and possible conversation paths within it, specifically it is the integration tests for the *botinterface* package that test "conversation cases" against the chatbot, for example:

```
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 represents all of 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 (the messages are piped through the BotRivescript object). 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 enginge) are also run. All such tests can be found between the *tests/tests_integration/test_botinterface* and *tests/tests_integration/test_concerns* folders.

Finally, the RiveScript Python macro code within *FlaskWebProject/chatbot/brain/python.rive* is not considered and not covered by tests and is in fact impossible to test in isolation. This is because of the fact that 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.

# Appendix C

# Categorizer Evaluation

The single-label categorizers used in the project where the two that are adapted from the nltk into the TextBlob package. A Naive Bayes classifier and a Maximum Entropy classifier.

The results of running the first against a test set (which is 10% the size of the complete dataset of 222 data points, around 22 test cases) is disappointing:

```
Accuracy: 0.36


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

```
Precision for 'family': 1.0
F for 'family': 0.57
Recall for 'emotion': 0.4
Precision for 'emotion': 0.4
F for 'emotion': 0.4
Recall for 'spiritual': 0.0
Precision for 'spiritual': 0.0
F for 'spiritual': 0

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

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

Recall for 'phys': 0.83              |Macro averaged recall: 0.4467
Precision for 'phys': 0.56           |Macro averaged precision: 0.4911
F for 'phys': 0.67                   |Micro averaged recall: 0.5454
Recall for 'pract': 0.4              |Micro averaged precision: 0.5454
Precision for 'pract': 0.4
F for 'pract': 0.4
```

```
Recall for 'family': 0.4
Precision for 'family': 1.0
F for 'family': 0.57
Recall for 'emotion': 0.6
Precision for 'emotion': 0.5
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 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 (see http://textblob.readthedocs.io/en/dev/_modules/textblob/classifiers.html#basic_extractor).

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 of the current project unfortunately left

no time to work on this aspect of the system.

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

# Appendix D

# Code Listing

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