

# **CHATBOT BASED MEDICAL EXPERT SYSTEM**

A project report submitted in partial fulfilment of the requirements for award of the  
degree

## **BACHELOR OF ENGINEERING**

In

## **COMPUTER ENGINEERING**

Of

## **SAVITRIBAI PHULE PUNE UNIVERSITY**

By

**JAIDEEP KEKRE**

**Exam No: B120234299**

**SAMEER DESHMUKH**

**Exam No: B120234250**

**BISHAL KUMAR**

**Exam No: B120234231**

**SANDEEP GAIKWAD**

**Exam No: B120234270**

Under the guidance of

**PROF. C. A. LAULKAR**



**Sinhgad Institutes**

**Department Of Computer Engineering**

**Sinhgad College of Engineering,**

**Vadgaon (Bk.), Pune-411041**

**2015-16**

# **CERTIFICATE**

This is certified that the project report titled

## **CHATBOT BASED MEDICAL EXPERT SYSTEM**

Submitted by

**Jaideep Kekre, Sameer Deshmukh, Bishal Kumar and Sandeep Gaikwad**

Have successfully completed their Project under the supervision of Prof. C. A. Laulkar for the partial fulfilment of Bachelor of Engineering in Computer Engineering of Savitribai Phule Pune University.

This work has not been submitted elsewhere for any degree.

**Prof. C. A. Laulkar**

**Guide**

**Prof. M. P. Wankhade**

**Head of Department**

**External Examiner**

**Prof. (Dr.) S. D. Lokhande**

**Principal**

## **ACKNOWLEDGEMENT**

We like to thank our faculty advisors Prof. C. A. Laulkar, Prof. N. G. Bhojane and Prof. S. S. Pawar for their guidance and patience without which this project would not have been possible.

We would also like to thank Dr. Manisha Deshmukh for her advice which helped us create the various disease and symptom databases and for helping in us get a good understanding of the caveats of medical history taking.

The judges and participants at the various college competitions where we presented our project provided valuable feedback which we incorporated into our project.

Numerous open source tools were used in this project, right from Python libraries to the Telegram app itself. The contributions of the hundreds of volunteers who have contributed toward making these projects what they are cannot be left unnoticed.

## **ABSTRACT**

As more and more research is done in various fields, the knowledge base (kb) keeps expanding incrementally and in some cases exponentially. Using such large knowledge base and inferring decisions based on such large data is complex and requires large amount of efforts and study. The decisions that are made on such existing knowledge are composed of logical statements.

We have built a system that uses existing knowledge in specific fields and assists the user in making decisions based on it. The system will converse with the user via a platform (Telegram Chat Bot) and provide knowledge based decisions.

The knowledge will be structured and decisions will be made upon this structure with inputs from user on what kind of decision or information is needed. Our system is highly modular and can be applied to any field. We have chosen the Medical field as it has wide ranging social benefits. The Medical field has a huge knowledge base of diseases, conditions, symptoms, medicines and seasonal conditions. This system will use the medical field as its knowledge base and attempts to simulate the first interaction that a patient has with a doctor.

This system does not and will not replace doctors, but will increase the ease of diagnosis and make the first visit of a patient to a doctor more productive. We have designed the system with the aim of improving the history taking experience and enabling patients to begin diagnosis remotely.

Our system consists of two major components i.e. Expert System and communication platform. We chose telegram due to its ease of use and bot friendly features. Each module that forms these components can be run independently and I designed to be swapped out if needed. We have used multiple processes instead of threads to get around the Python Global Interpreter Lock. We also use an in-memory database called redis which is an R.A.M based key value store. The Expert system is run by 3 major and two minor algorithms which we have created from scratch for this system.

Our system's algorithms have successfully asked the most optimum questions to the user during the consultation and yielded upwards of 80% probability of having a disease in tests. Our system can generate reports for doctors based upon patient data and assign probability score to each patient's susceptibility.

## LIST OF FIGURES

<b>Sr. No.</b>	<b>Figure Name</b>	<b>Pg. No.</b>
3.1	System Architecture Diagram	22
3.2	Use-Case Diagrams	23
3.3	Class Diagram	24
3.4	Patient State Transition Diagram	25
3.5	Deployment Diagram	26
3.6	Patient Interaction Sequence Diagram	27
4.2	Telegram GUI	32
4.3	Disease Diagram	41
6.1	Main GUI Snapshots	44
6.1(a)	GUI 1	44
6.1(b)	GUI 2	44
6.1(c)	GUI 3	44
6.1(d)	GUI 4	44
6.1(e)	GUI 5	44
6.1(f)	GUI 6	44
6.1(g)	GUI 7	45
6.1(h)	GUI 8	45
6.1(i)	GUI 9	45
6.2(a)	Server Snapshot 1	45
6.2(b)	Server Snapshot 2	46
6.2(c)	Server Snapshot 3	46
6.3	Redis Database Dump	47

## LIST OF TABLES

<b>Sr. No.</b>	<b>Table Name</b>	<b>Pg.No.</b>
2.1	Cost and Effort estimates	17
2.2	Value of each function	17
2.3	Time 1	18
2.4	Time 2	18
2.5	Time 3	19
2.6	Time 4	19
2.7	Time 5	19
2.8	Time 6	20
2.9	Time 7	20
2.10	Time 8	21
2.11	Time 9	21

## **ABBREVIATIONS**

- AI – Artificial Intelligence.
- API – Application Programming Interface

## NOMENCLATURE

- Telegram – Telegram is a cross platform instant messaging application that lets users send messages from a suitable device employing Android, iOS, Windows Phone, OSX, Windows Desktop or even a simple browser.
- Telegram command – A telegram command is a word that is preceded by a ‘/’ symbol. This word then becomes a special command for a chat bot that is at the receiving end of the conversation in progress. Users can specify special commands like getting help or specifying their username by clever usage of telegram commands.
- Telegram command handler – A telegram message handler is a function in a chat bot that handles special telegram messages sent to it by users, processes it according to some predefined logic and then replies back to the user based on that logic.
- Telegram message – A telegram message is a text message that can be sent either by a user or a chat bot.
- Telegram Message Handler – This is similar to a telegram message handler, the only difference being that a message handler is made for processing all sorts of messages, not just commands.
- Expert System – Expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented primarily as if–then rules rather than through conventional procedural code.



## TABLE OF CONTENTS

<b>Title page</b>		<b>i</b>
<b>Certificate page</b>		<b>ii</b>
<b>Acknowledgement</b>		<b>iii</b>
<b>Abstract</b>		<b>iv</b>
<b>List of Figures</b>		<b>v</b>
<b>List of Tables</b>		<b>vi</b>
<b>Abbreviation</b>		<b>vii</b>
<b>Nomenclature</b>		<b>viii</b>
<b>Chapter No.</b>	<b>Title</b>	<b>Page No.</b>
<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
1.1	Background and basics	1
1.2	Literature Survey	2
1.3	Project Undertaken	3
	1.3.1 Problem definition	3
	1.3.2 Scope Statement	3
1.4	Organization Of Project Report	3
<b>2</b>	<b>PROJECT PLANNING AND MANAGEMENT</b>	<b>4</b>
2.1	System Requirement Specification	4
2.2	Project Process Modelling	13
2.3	Cost & Efforts Estimates	15
2.4	Project Scheduling	16
<b>3</b>	<b>ANALYSIS &amp; DESIGN</b>	<b>20</b>

3.1	System Architecture Diagram	20
3.2	Use-Case Diagrams	21
3.3	Class Diagrams	22
3.4	State Transition Diagrams	23
3.5	Deployment Diagrams	24
3.6	Sequence Diagrams /DFD & CFDs	25
<b>4</b>	<b>IMPLEMENTATION &amp; CODING</b>	<b>27</b>
4.1	Database schema	33
4.2	GUI Design	35
4.3	Operational Details	37
<b>5</b>	<b>TESTING</b>	<b>40</b>
5.1	Acceptance Testing	40
5.2	Unit Testing (module wise testing)	40
5.3	Integration Testing	41
<b>6</b>	<b>RESULTS &amp; DISCUSSION</b>	<b>42</b>
6.1	Main GUI snapshots	42
6.2	Terminal snapshots	43
<b>7</b>	<b>CONCLUSION</b>	<b>44</b>
	<b>References</b>	<b>45</b>
	<b>Appendices</b>	<b>48</b>

## Chapter-1

### Introduction

---

The aim of this project to develop a chat bot based medical expert system. Doctors in India have to work long hours and see hundreds of patients a day. A lot of time is spent by doctors for gathering the personal and medical history of patients with respect to their demography and present and past symptoms.

The objective of the project is to enhance the first doctor-patient interaction where the doctor asks the patient certain questions for noting down their history. We believe that a patient's basic history taking can be done by having a conversation with a friendly chat bot, so that the doctor is freed from this activity and precious time is saved.

This project will only aim to act as a middle man between the patient and doctor. It works by letting the patient chat with the system so that he/she can report their personal and medical history to the computer, which the computer keeps a track of. The system will then try to infer a diagnosis from this interaction and report it directly to the doctor by sending the doctor an email.

#### 1.1 Background and Basics

Expert system is a computer system that emulates the decision-making ability of a human expert. As described above, most doctors in India are over-burdened by the number of patients that they have to see on a daily basis. We have come up with a novel solution to this problem by creating a chatbot based medical expert system that can have a friendly conversation with a patient for making note of their medical history.

For this purpose we have used the Telegram messaging app as a front-end for the patient. Telegram is a widely used, cross-platform, open source and highly secure chat application that lets bots have conversations with users. It does this through a RESTful API which any chat bot can interface with.

## 1.2 Literature Survey

Chat bot based expert systems have existed since the past 30 years. A notable example is the famous 'Eliza' chat bot.

Ontology based chatbots are an on-going effort that relies on the semantic web and NLP models to support user-machine dialogical interactions. The system works by using 3 modules - a knowledge based system, question interpretation, and Natural Language Generation. Ontology is a formal naming and definition of the types, properties and inter-relationships between entities.

An important development in creating chat based expert systems is the development of Temporal Reasoning systems. Temporal reasoning is important because the final diagnostics depends heavily on the sequence of the symptoms. Temporal reasoning and logic is a system of rules for representing and reasoning about propositions qualified in terms of time for example I am ALWAYS hungry. The paper explores temporal relationships between symptoms based on fuzzy set theory.

Another paper surveyed reveals interesting ways of modelling the medical decision making process. It reviews AI and expert systems to acquaint the reader with the field and to suggest ways in which this research will eventually be applied to advance medical monitoring. It states that an expert system is a system that contains and can apply specialized knowledge. It uses this knowledge to make suggestions to users who may not have the full range of expertise available.

Kisileva and Toropchina's paper analyses and precisely defines the functioning of a medical expert system. It says that an expert system is a program that acts similarly to an expert in some subject field. The difference between expert systems and knowledge based systems is that an expert system is able to explain its behaviour to a user. The paper also analyses different approaches taken for implementing some medical expert systems.

We also went through a paper by Hill, Ford and Farreras that analyses how the quality of conversations changes between human-human and human-chatbot interactions. They conclude that while human communication skills translate easily to a computer, there were many differences between the quality and the content of these conversations. It was found that most human-chatbot interactions were short and lacked richness of vocabulary, and usually involved more profanity.

## **1.3 Project Undertaken**

### **1.3.1 Problem Statement**

Development of a chat bot based medical expert system to automate the history taking procedure of patients and in effect enhance the first doctor patient interaction by giving the doctor access to a patient's medical history and possible diagnosis without having to see the patient personally.

### **1.3.2 Problem Scope**

What's in:

1. Let's patients chat with the system to extract an accurate history from them.
2. Let's doctors see the medical history of their patients through a similar chat interface.
3. Tries to zero in on the disease that the patient might have by relating symptoms to each other.

And what's out -

1. Does not aim to replace doctors.
2. Cannot detect and accurately extract symptoms of complex diseases like cancer or diseases where manual check-up or tests are essential.
3. Limited by the knowledge base.

## **1.4 Project Outline**

In this document we have elaborated on how we went about implemented our chat bot based medical expert system. It encapsulates the core algorithms, data base schemas and overall functionality and caveats of all the modules.

At the end of this document we have provided an overview of the testing methodologies that we have used and the results obtained from real world testing.

## Chapter 2

### Project Planning and Specification

---

#### 2.1 Software Requirement Specification

##### 2.1.1 Overall Description

###### 2.1.1.1 Product Perspective

Expert systems have been around for a very long time, but their application to medical history taking is new. This system will be developed from the ground up by taking inspiration from various expert systems and chat bots made in the past.

The chat bot based medical expert system will chat with a patient and take a note of his symptoms. These symptoms will then be transferred to a doctor for further analysis.

This system is a tool for aiding doctors in history taking so that the time that is often spent in this task can be better utilized.

###### 2.1.1.2 Product Functions

Product functions from the perspective of the users can be written as follows:

- The primary user interface for patients is text based chat.
- Users will interact with the system through the Telegram app which lets them enter text and send it to a web server.
- There are two primary user's patients and doctors. Patients will chat with the system to report their disease symptoms and the system will respond in such a way so as to incite a response for further probing symptoms. The objective of the system will be diagnose the disease that the patient is suffering from, and also collect relevant data about other personal parameters like age, gender, height and weight.

- The other users of the system will be doctors. They will interact with the system through email. Emails detailing the symptoms and possible diagnosis of patients based on the collected information will be sent to doctors once a patient successfully completes a chat with the system.
- A third user of the system is the admin. The role of the admin is to authorize a doctor to view the details of a particular patient. Since medical history is of a very personal nature, only the admin (who will be a trusted person of the company deploying this service) will be able to set permissions for doctors. The admin will be able to interact with the system through a command line application.

### **2.1.1.3 User classes and characteristics**

**There will primarily be 3 classes of users:**

#### **Patients**

- The primary users of the system. Patients will chat with the system through a chat interface.
- They will communicate their symptoms to the system and the system will ask them relevant questions to figure out a diagnosis.
- They will chat with the system and engage in a chat for determining the exact disease. Patients are the most important users of the system.

#### **Doctors**

- The doctors will use the system for accessing data about patients.
- They will be able to view all the patients whose data they have access to and also view specific details about each patient.

#### **Admin**

- The admin will be responsible for maintenance and upkeep of the system.
- The admin will be sole person who will be able to set permissions for doctors who would want to access data of a given patient.

#### 2.1.1.4 Operating Environment

Operating environment will be as follows:

- Any UNIX based operating system. We will be using Debian/Ubuntu.
- Internet connection.
- Python interpreter.
- Various python libraries like python-telegram-api, threading, multiprocessing, etc.
- Redis.
- Computer running commodity hardware.

#### 2.1.1.5 Design and Implementation Constraints

A major concern is maintaining the privacy of user data. Only specific doctors should be able to access the symptoms related data of certain patients and that too with their consent. Admin access to this data will be restricted to the chief maintainers of the system. Thus data will have to be stored in a very secure manner and it cannot be distributed without specific legal permission from patients.

The front end of the system (user facing) will depend on the Telegram mobile app. We will need to conform to the standards set by Telegram for chat communication between client and server. All the server side code will be written in Python and the database will be Redis. We will primarily use HTTP get and post requests to send and receive data over the network.

The students working on the project will be its sole maintainers. No 3rd parties involved.

#### 2.1.1.6 User Documentation

The client side chat bot is designed to be self-explanatory. Instructions on using the chat bot will be delivered to users through the chat interface whenever necessary. There will be no external documentation for users as such.

#### 2.1.1.7 Assumptions and Dependencies

All the tools being used for the project are released as Open Source Software and are freely available over the internet. The front end depends on the Telegram mobile app, which is also on its way to being released as open source software.



We also assume that users are connected to the internet at the time of initiating conversations with the chat bot.

### **2.1.2 External Interface Requirements**

#### **2.1.2.1 User Interfaces**

##### **Interface between patient and system:**

- This interface is already created by Telegram, and we will be using the same to take information from the patient and deliver the relevant information back to them.
- It predominantly features a text box for input of text, a chat window that displays previous conversations, a keyboard for typing responses and a 'send' button for sending this information to the system.

##### **Interface between doctor and system:**

- The only information that the doctor expects from the system is information about his/her patients. This will be delivered to them via their email account.

##### **Interface between admin and system:**

- The admin takes care of assigning doctors to patients. This will be done through a command line interface.

#### **2.1.2.2 Hardware Interfaces**

We envision the primary interface between the patient and the system and the doctor and the system to be a smartphone which will be able to run the Telegram app. Thus any smart phone that can run relevant versions of Android, iOS or Windows Phone will be a contender for supporting chats with the user. The smart phone should support internet connectivity and should be compatible with the latest HTTP standards.

Doctors will of course need a similar internet connected computer and an email account.

### 2.1.2.3 Software Interfaces

The project consists of two major parts, one that resides on the client side (chat interface) and that other that resides on a server (chat bot). The client side part of the system relies on the Telegram mobile app, which must establish a 2 way communication with the server side software for the system to work properly. There is also a data base involved in case of the chat bot. Most of the software interface requirements stem from this architecture. They are divided into 2 categories, those that are used by the client and those used by the server:

#### Client side software interfaces:

- Operating system Android (v2.2 and up) or iOS (v6.0 or later) or WindowsPhone (v8 or v10).
- Telegram v3.2 or later (Android), v1.15 or later (Windows Phone), v3.2.1 or later (iOS).
- HTTP network stack (TCP/IP).
- The client side will send text messages packaged into HTTP packets to the server and will receive text and Telegram keyboard layouts as responses.

#### Server side software interfaces:

- Internet connected and HTTP ready UNIX instance, either standalone or in the cloud.
- Redis.
- Various python libraries like python-telegram-api, threading, multiprocessing, etc.
- Server receives data in the form of HTTP packets that will contain JSON, which will have the text message and credentials of the user.
- It sends out HTTP packets containing JSON, which contain details of the response from the chat bot.

### 2.1.2.4 Communications Interfaces

- Smart phone supporting HTTP communication protocols.
- Server capable of receiving and sending HTTP messages.

### 2.1.3 System Features

#### 2.1.3.1 Chat Bot

##### 2.1.3.1.1 Description and Priority

This is the heart of the project, having highest priority of 9. It is the core of the project that will interact with the patient in a natural language. It will have provisions to accept text input from the user, process it, compose a reply and send it over the network.

High risk component. Risk rating 9. This the sole method of input for the patient; any error here will show up in the entire system.

##### 2.1.3.1.2 Stimuli - Response Sequences

###### Patient chats with the system:

- **Stimulus:** Patient sends a text message describing his ailment.
- **Response:** Chat bot pulls the user's details from the database into active memory for further processing. System will initiate a chat and try to diagnose the patient's disease. This will involve read/write calls to the data base and business logic modules. Once the chat reaches a 'finished' state, an appropriate message will be sent to the user indicating this.
- **Stimulus:** Patient replies with the Telegram in-app keyboard to the question asked by the system.
- **Response:** The chat bot records this response, and after processing is done by the business logic module, an appropriate reply is sent back to the user. This might be another question to prompt the patient for more information or a message announcing that the conversation is over.

###### Doctor-system interaction:

- **Stimulus:** A patient whose data has been authorised to be viewed by a particular doctor has just successfully completed a conversation with the system.
- **Response:** The system will fetch the relevant patient details about the patient and return it to the doctor in a readable and understandable format. The information will be sent to the doctor as an email. The responses will typically concern obtaining more information about a particular patient.

**Admin-system interaction:**

- **Stimulus:** Admin wants to authorize a doctor to be able to receive information about a given patient.
- **Response:** System associates doctor credentials with the patient and authorises doctor to access patient information.

**2.1.3.1.3 Functional Requirements**

## 1. Server module

- This module is responsible for receiving and sending messages by interacting with the Telegram server. Additionally it is also responsible for dispatching messages to the expert system modules and getting replies from them.

## 2. Database communication module

- This is the module that is completely responsible for read/write calls to the database. The data base module will supply the relevant user data to the chat bot and receive data to be stored in the data base

## 3. Dispatcher module

- The dispatcher acts like a router. It allocates a single object to an independent conversation.

## 4. Question and response data store

- This is a static data store which stores the data about questions and responses.

## 5. Core module

- This module is a state aware module that is spawned on a per conversation basis. It is aware of the state that the conversation is in at any point of time.

## 6. Expert System module

- The Expert System module sits between the core and Doctor Sky Net modules. It takes care of handling conversation states and communication between Core and Doctor SkyNet.

## 7. Doctor SkyNet module

- This is the core module that contains various crucial algorithms for actual functioning of the entire system.

### 2.1.3.2 Chat Interface

#### 2.1.3.2.1 Description and Priority

This is second most important part of the project apart from the chat bot itself. It can safely be assigned a **priority of 7**. The chat interface will be the users' gateway into the system. Specifically, we will be using the 'Telegram' mobile app for this purpose. Telegram lets us interface a chat bot that can send and receive messages to users. One of its major features is that it lets us show the user a custom keyboard, which can contain only specific buttons. This is helpful for restricting the user's input to very specific replies and exercise considerable control over the interaction between the user and the chat bot.

#### Major functionalities:

- Accept raw text entered by the user.
- Convert the received text to JSON and transmit it over a network to the chat bot.
- Receive a text message reply from the chat bot and present it to the user.
- Receive a layout for a custom keyboard from the chat bot and present it to the user.
- Accept a reply on this custom keyboard and text and transmit that to the chat bot.

#### Risks:

- Most of the risks associated with this part of the project stem from the Telegram app itself. For example, what if the company behind Telegram decides to disallow interfacing of chat bots? We'll need to think of an alternate way to allow for the chat interface then.
- Another risk is the loss of internet connectivity.
- But none of these risks would be fatal to the project since the chat bot is designed to be agnostic of the front end (in this case Telegram). It can easily adapt to a web interface or even a custom mobile app.
- Considering these factors, a risk factor of 6 has been assigned to the chat interface.

#### 2.1.3.2.2 Stimuli - Response Sequences

Patient wants to start conversation

- **Stimulus:** Patient or doctor expresses his intent to start a conversation by opening the Telegram app and start typing something into the text field.
- **Response:** The interface waits until the user hits the 'send' button present on the app. Once this is done the text entered along with his/her user credentials is packaged into an HTTP packet and send over the network to the chat bot.

Patient conversation already in progress

- **Stimulus:** Patient or doctor receives a message from the chat bot through the chat interface prompting him/her to enter a reply. The user has to reply through the custom keyboard and is able to send only specific replies.
- **Response:** The chat interface sends this response as an HTTP packet to the chat bot, where the chat bot can do its job.

#### 2.1.3.2.3 Functional Requirements

- An internet connected smartphone running Android, iOS or Windows Phone. The internet serves as bridge between the user facing (chat interface) and data processing (chat bot) modules of the project. One of the above 3 operating systems is required on the smart phone since Telegram is heavily dependent on them.
- The Telegram app installed on the smartphone.

#### 2.1.4 Other Non-Functional Requirements

##### 2.1.4.1 Performance Requirements

Performance mainly concerns the users being able to receive a response for their input within a reasonable amount of time. This has been set to around 5 seconds for a chat response to leave the system. We do not consider the delivery time of the message here because that solely depends on the user's network speed. Thus the algorithms and hardware used in the system should ensure that a response leaves the system within 5 seconds of receiving a user's message.

#### **2.1.4.2 Safety Requirements**

Great safety must be maintained when handling and securing the data collected from patients. It is particularly important to secure access to this data such that it can only be accessed by the doctors who are associated with the patient. This concern mainly arises out of the legal and ethical restrictions that arise when dealing with data as personal as a person's medical history.

While we will try our level best to ensure that the system is error proof for all the tasks that it is programmed to do, it might throw up wrong output that might arise from malfunctions in the core algorithms of the system (present in the Expert System and Doctor SkyNet modules). Thus the concerned doctor must verify the output that he receives from the system with his own domain expertise.

#### **2.1.4.3 Security Requirements**

It is of utmost importance to completely ensure that the patient's personal data that will be collected and stored by the system is highly secure and no unauthorized access takes place with this data.

Since the front end being used is the Telegram app, the unique username from Telegram will be used to authenticate users.

#### **2.1.5 Other Requirements**

##### **2.1.5.1 Legal Requirements**

Strict medical ethical and legal practices make it mandatory for us to safeguard patient data and only reveal it to doctors who have been authorized to access the data of a particular set of patients. Thus all such requirements will be enforced in the system.

## **2.2 Project Process Modelling**

Incremental Process model will be suitable for implementation of this project. Because, we are planning to start implementation from identifying simple diseases and symptoms and then subsequently increase the kinds and complexity of the diseases involved. We expect to deliver iterations of software with improvement in performance or adding new feature in each iteration.

## Framework Activities –

1. Survey of research and communication: This activity involves surveying current research and existing approaches as well as communicating with project guides, review panel members for refinement of project idea.

### Task Sets

- **Tasks:** Conduct research on idea and study existing approaches. Evaluate pros and cons. Collect relevant journal/research papers. Explain idea to project guides and note their comments.
  - **Deliverables:** Comparison of all existing approaches and selection of methodology to use in the project.
2. Planning: Planning the project involving timeline preparation, technical tasks to be done, feasibility and risk analysis etc.

### Task Sets

- **Tasks:** Division of implementation activities across the given time for project, assigning group members responsibilities, obtaining the programming resources and tools needed to implement project.
  - **Deliverables:** Risk chart, feasibility analysis using mathematics, selection of emulator for the gaming console, Project timeline chart.
3. Design and Modelling: The system to be developed is to be designed using UML diagrams and mathematical modelling of the system is to be done.

### Task Sets

- **Tasks:** System breakdown to identify components, identify actors in system, features to be present and not present, interaction between components, flow of data, identifying classes.
  - **Deliverables:** System architecture diagram, Component diagram, Use cases, Class diagram, flowchart.
4. Coding: Coding of components of system, binding components together and testing is performed.



## Task Sets

- **Tasks:** Coding of components by respective group members as per the design approved in previous activity, Interfacing components and testing.
- **Deliverables:** Core software implementation in first iteration, result of testing features, identifying bugs and failures if any.

### 2.3 Cost and Efforts Estimates

Using Function point calculation, the estimate of software size is evaluated as follows. Function points are basic data from which productivity metrics could be computed.

FP data is used in two ways:

- As an estimation variable that is used to 'size' each element of the software.
- As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

Table 2.1 Cost and Effort estimates

Info Value Domain	Optimistic	Realistic	Pessimistic	Est. Value	Wt. Factor	FP
No. of Inputs	5	5	12	6	3	18
No. of Outputs	5	7	10	7	4	28
No. of Files	2	5	8	3	1	3
No. of External Interfaces	3	4	6	4	2	8
						Total = 57

Table 2.2 Value of each function

Sr. No.	Factor	Value
1	Backup and Recovery	2
2	Data Communication	0
3	Distributed Processing	0
4	Performance Critical	5
5	Existing Operational Environment	3
6	Online Data Entry	0
7	Input Transactions over multiple screens	0
8	Online Updates	0
9	Complex Information Domain Values	3

10	Complex Internal Processing	3
----	-----------------------------	---

Complexity Adjustment Factor =  $[0.65 + (0.01 * 24)] = 0.156$

FP estimated =  $(57 * 0.156) = 8.892$

## 2.4 Project Schedule

The schedule of our project is as follows:

### June

Table 2.3 Timeline 1

Date	Activity Planned	Activity Completed
22nd June 2015	Initial planning phase and project quantization.	Planning done, quantization complete.

### July

Table 2.4 Timeline 2

Date	Activity Planned	Activity Completed
2 July 2015	Figuring out creation of module diagram and delegating responsibilities.	Successfully understand creation of module diagram.
6 July 2015	Discuss references from published papers with respect to our project.	Extended sources discussed and their relationship finalized with our project.
13 July 2015	Finalize problem statement & prepare for 1 <sup>st</sup> review.	Problem statement finalized & presentation planned.
25 July 2015	Successfully clear 1 <sup>st</sup> review.	First review cleared and due feedback acknowledged.

**August**

Table 2.5 Timeline 3

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
3 August 2015	Gain a basic understanding of data flows & mathematical models.	Understood and initialized certain data flows and caveats of mathematical model.
17 August 2015	Submit initial mathematical model and get feedback.	Initial model submitted. Much feedback received.
24 August 2015	Make a better mathematical model & get more feedback to improve on it.	Received due feedback on new Mathematical model.

**September**

Table 2.6 Timeline 4

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
5 Sept 2015	Discuss and understand the functioning of UML diagrams.	Finished as planned.
12 Sept 2015	Discuss & understand the role & purpose of the software required specification.	Finished as planned.
19 Sept 2015	Prepare both SRS & UML diagrams and get them verified.	Finished as planned.
26 Sept 2015	Get an idea of project modules based on case studies and finalize them.	Project modules verified. Use case approved.

**October**

Table 2.7 Timeline 5

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
3 Oct 2015	Make a mock program & presentation and prepare for 2 <sup>nd</sup> review.	Preparation done & mock program modules verified.

5 Oct 2015	Present the second review successfully.	Second review finished with flying colours.
17 Oct 2015	Familiarize ourselves with different python libraries that will be used.	Libraries understood final and shortlisted and confirmed with guides.
24 Oct 2015	Begin with a very basic implementation of telegram chat bot.	Telegram bot registered. Mock publish submit bot created.

## December

Table 2.8 Timeline 6

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
26 Dec 2015	Define system architecture and schemes for creating a basic chat bot.	Finished as planned.
31 Dec 2015	Modules in code and make basic pub sub telegram bot for echoing input.	Demo for very basic telegram bot ready.

## January

Table 2.9 Timeline 7

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
9 Jan 2016	Implement a multithreaded server for receiving & returning messages.	Server implemented works properly in asynchronous manner.
16 Jan 2016	Create the dispatcher and core classes and couple them with server.	Dispatcher and core completed and coupled with server.
23 Jan 2016	Create schemas for knowledge bases automate testing provision creation.	Knowledge base schemas ready. Automated testing provision made for each module.
30 Jan 2016	Interface for telegram keyboards Create question interface class and begin populating	Telegram keyboards interface finished. Question interface created with associated helpers.

	knowledge base.	
--	-----------------	--

**February**

Table 2.10 Timeline 8

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
6 Feb 2016	Create symptom validity tables & scratch pad classes add database helpers.	Scratch pad finished. Database connection secured & helpers coded.
9 Feb 2016	Show a working demo of basic question answer based chat bot for 3 <sup>rd</sup> review.	3 <sup>rd</sup> Review completed
20 Feb 2016	Formalize schemas for disease interface and algorithm for expert system.	Algorithms planned. Schemas formalized.
27 Feb 2016	Finish coding of disease interfaces make and test expert systems algorithms.	Finished as planned.

**March**

Table 2.11 Timeline 9

<b>Date</b>	<b>Activity Planned</b>	<b>Activity Completed</b>
5 March 2016	Advance disease detection algorithms & make them generic.	Disease detection working with buckets support. Few disease data samples added too.
12 March 2016	Make AI generic & introduce DoctorSkyNet.	Finished as planned.
19 March 2016	Have a fully functional medical expert system ready.	Medical expert system is ready. Chat bot converses with users.
26 March 2016	Add more data for diseases symptoms and questions.	Project finished.

## Chapter 3

### Analysis and Design

#### 3.1 System Architecture Diagram

The Architecture Diagram of the system shows the whole system's modules and the interactions between them at a glance. Notice that the User never directly comes in contact with the Doctor – all sorts of interactions between Doctor and User always happen through the chat bot.

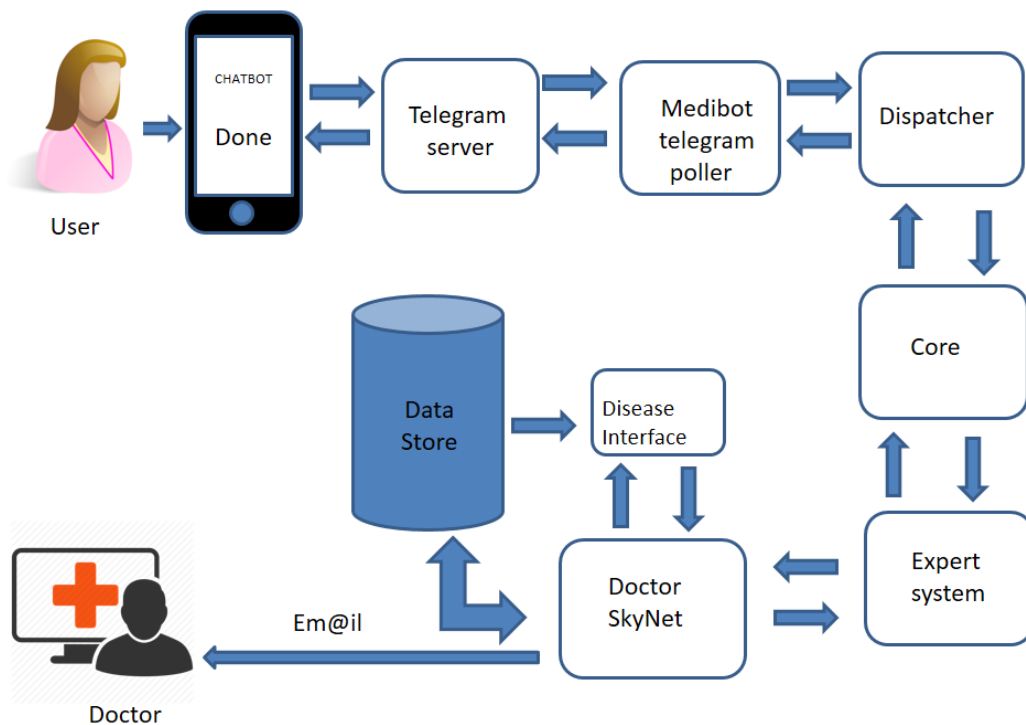


Fig 3.1 System Architecture Diagram.

A diagram showing the architecture of the entire system as a whole, along with the interaction that takes place between different modules. The User initiates a conversation with the system by sending a message, which is captured by the Telegram Server and then sent across through the different modules of MediBot. A conversation with the User is initiated, which then eventually terminates in the Doctor getting an email from the chat bot about the User.

### 3.2 Use-Case Diagrams

Use Case diagram shows the interaction between different actors and modules of the system. In this case you can see the most important modules of MediBot and the actors that they interact with.

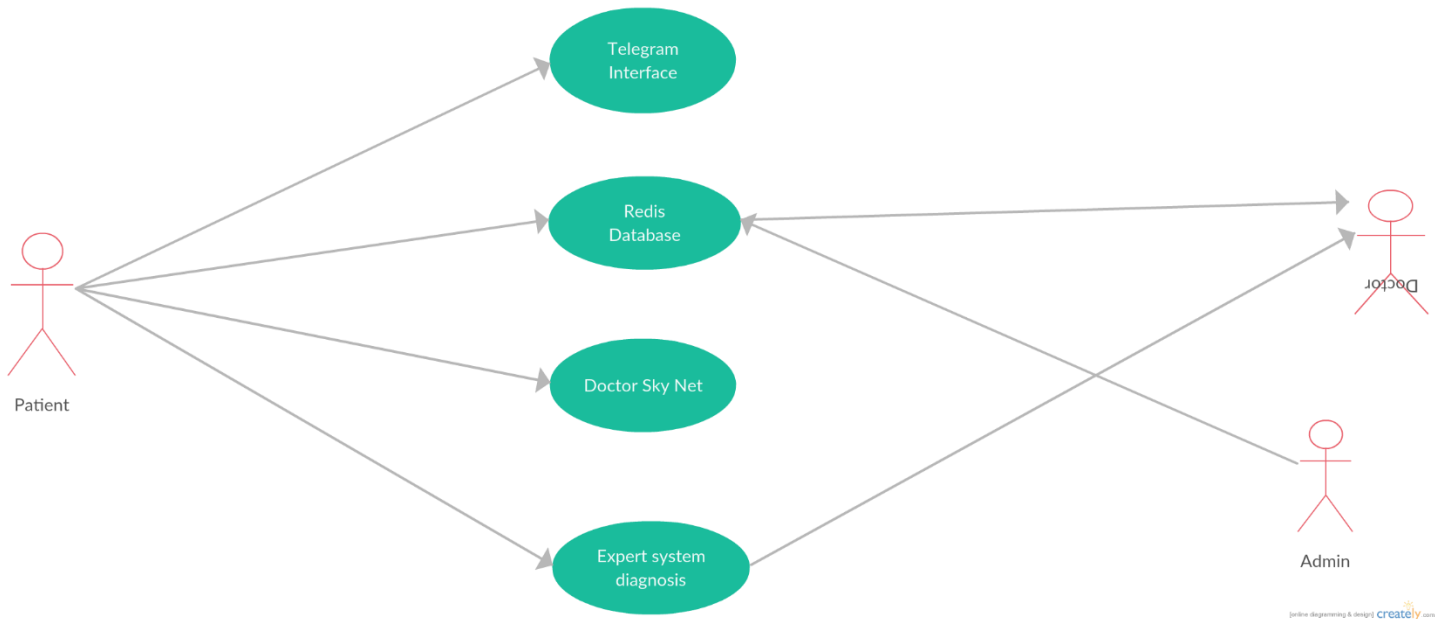


Fig 3.2 Use Case Diagram.

The main modules of the system that actors interact with are the Telegram Interface, Redis Database, Doctor SkyNet and the Expert System Diagnosis. The main actors are the Patient, Doctor and Admin. The Patient is the most important actor of the system without whom the system would not function since a chat will never instantiate. The Patient interacts with all the modules since the normal flow of a conversation makes the use of all modules highly essential. The Doctor's role with respect to the system is limited to obtaining information about the Patient which he can consume later as per his/her requirement. Thus the Doctor only receives information from the Redis and Expert System Diagnosis modules. The Admin's only job is to monitor system usage and assign access privileges to Doctors, hence the Admin only needs to interact with the database, whenever he needs to change any information.

### 3.3 Class Diagrams

A class diagram of the system showing the interaction between various classes and data flows between them. Each class contains behaviour and attributes and one-to-one or one-to-many associations.

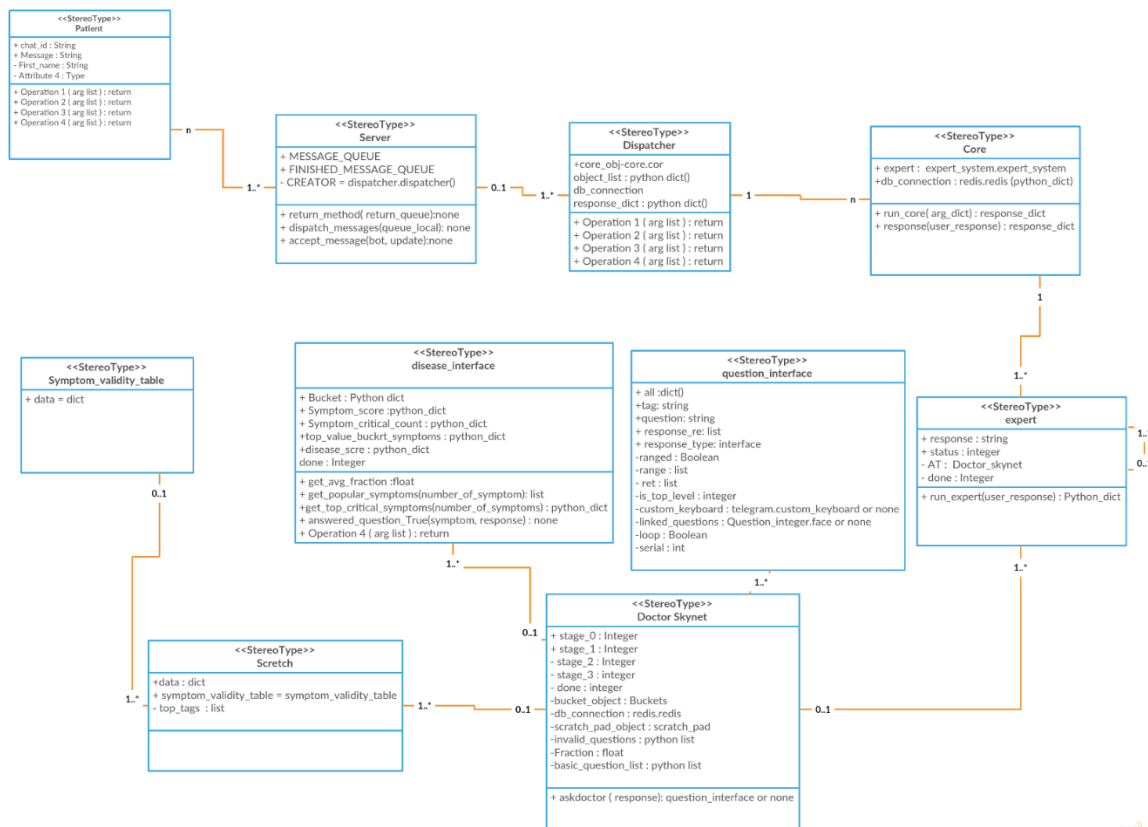


Fig 3.3 Class Diagram

The class diagram consists mostly of the classes and modules that have already been specified in the System Architecture diagram. In addition to those, certain classes which are essential to realize the functionality of the core classes have also been specified here. For example, the **question\_interface** and **disease\_interface** classes are essential for making the **DoctorSkyNet** class work since questions that are to be asked to the patient and an interpretable form of the diseases that the patient might have are an important part of determining the next question that is to be asked to the user.



### 3.4 State Transition Diagrams

The Patient State Transition Diagram denotes the transitions in the state of the Patient that take place as he/she proceeds to have a conversation with the system. The states (boxes) denote the activity that the Patient performs in a particular state and the arrows denote the action that the Patient must take in order to reach that state.

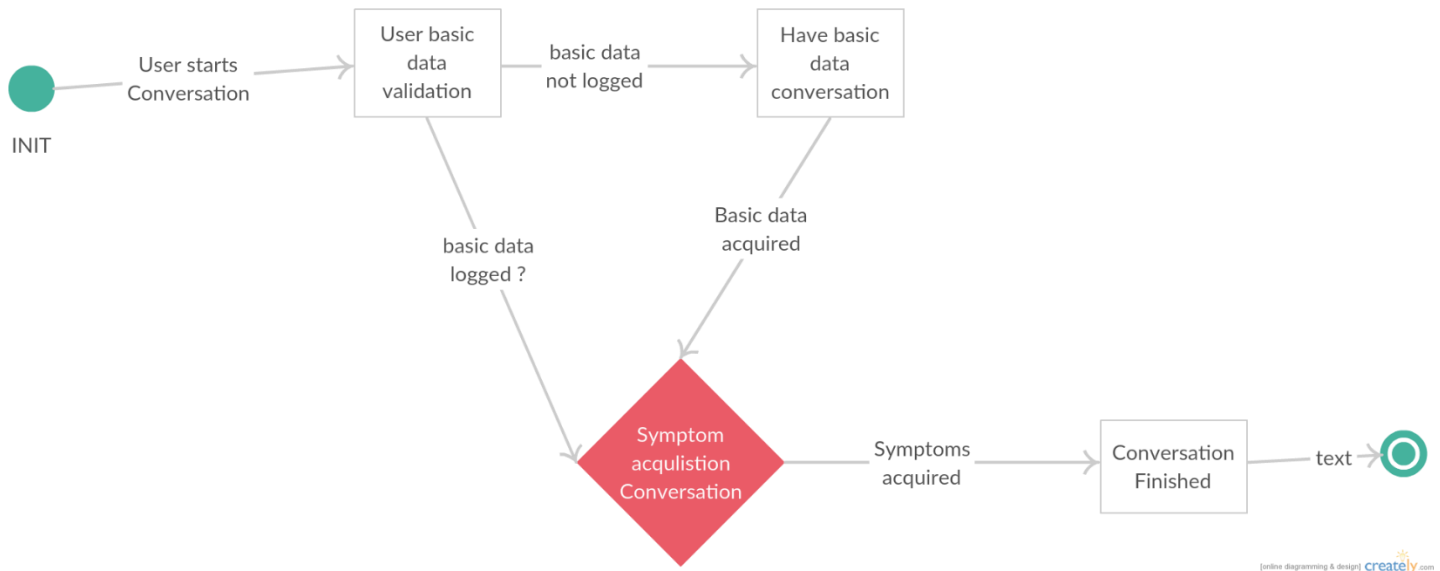


Fig 3.4 Patient State Transition Diagram.

The above state transition diagram denotes an overview of the actions that a Patient can perform and the actions that he must take in order to go from performing a set of actions to another set of actions. As per the diagram it is very clear that the patient must first validate his basic data in order to proceed to have a conversation with the system that will allow him/her to report their symptoms to MediBot. The state diagram makes it clear that the system will first have a conversation for acquiring basic data from the user before it even allows them to start a Symptom Acquisition Conversation. Once the symptoms have been successfully acquired, the conversation can be terminated.

### 3.5 Deployment Diagrams

Diagram showing how the system will be deployed on a real computer system. The deployment diagram helps us understand precisely the conditions under which the entire system will be deployed, and lets us see the way the in which different components can integrate with each other in the real world.

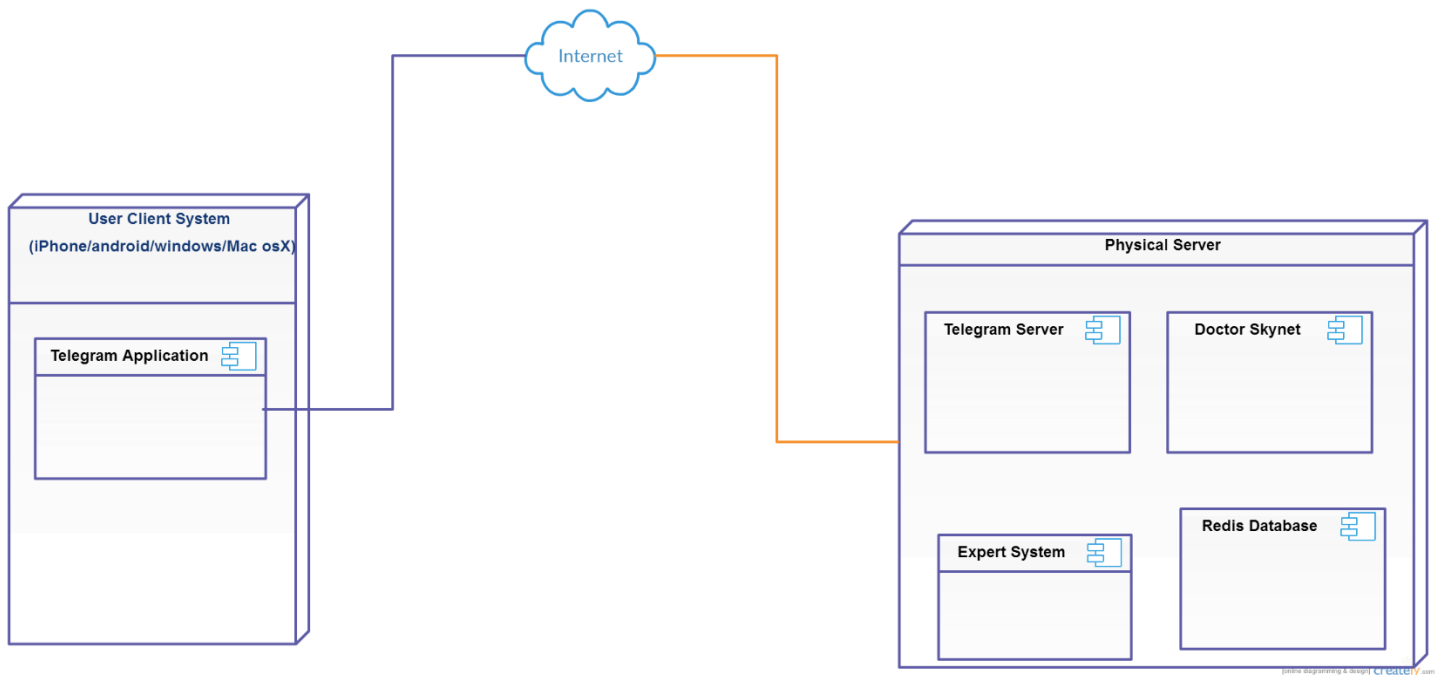


Fig 3.5 Deployment Diagram.

As can be seen in the diagram above, the system consists primarily of two deployments, the User Client System that runs the Telegram chat application, which can be anything from an iPhone to a Windows PC, and the Physical Server, that actually runs the chat bot by accepting user input from the Telegram app. Both these deployment areas are connected by means of the Internet. As has been described in detail in this document, the Physical Server can be any UNIX machine and the User Client System must be any environment that supports the Telegram chat application.

### 3.6 Sequence Diagrams

The sequence diagram shows the time based actions of user with respect to the modules of MediBot. The major modules consist of the Telegram Application, Server and Expert System.

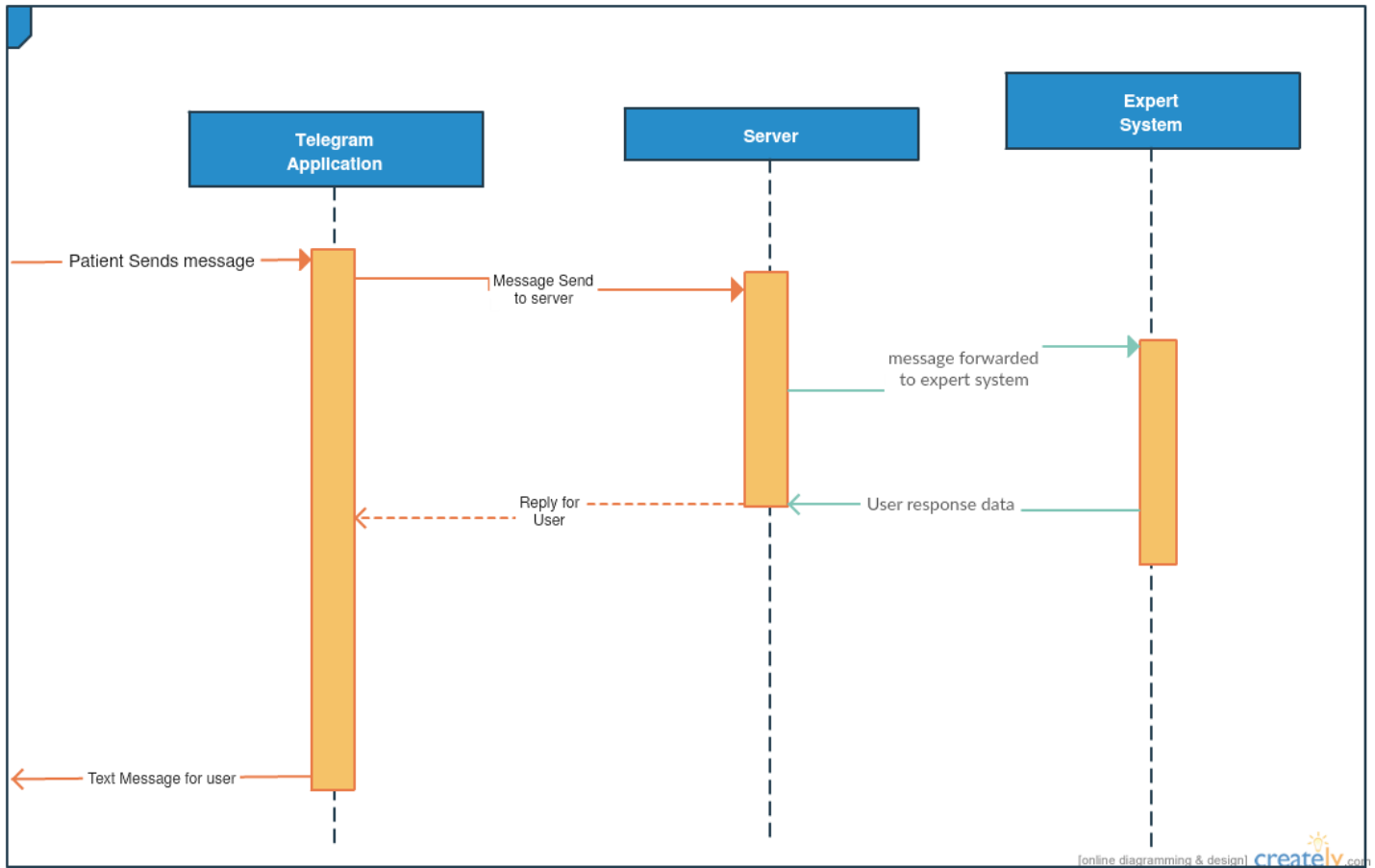


Fig 3.6 Patient Interaction Sequence Diagram.

As can be seen in the above diagram, the patient must first interact with the Telegram Application in order to initiate a conversation with the system. The Telegram Application then packages the message in its own format and sends it over to the Server, which then sends it to the Expert System. The Expert System is responsible for gaining a deeper understanding of the message in context with other messages that have been sent to the chat bot previously and according to the user's already saved data if any. It generates a suitable response for the sent message based on these parameters and sends it across through the Server and Telegram Application again so that it can reach the user.

## Chapter 4

### Implementation and Coding

---

#### 4.1 Database schema

The system as a whole relies on two kinds of data –hard coded data and dynamic data. They are represented as follows:

##### 4.1.1 Hard coded data

- This data either pertains to symptoms, diseases or questions. It highly important data that must be in a particular format. It is the primary knowledge store of the expert system.
- It is stored as Python native data structures like dictionaries, lists, strings and numbers – all in dedicated python files. The reason for keeping these hard coded and not in any external database is to completely nullify the possibility of unwanted elements making changes to the data, which might prove disastrous for the system as whole.
- The two major types of hard coded data are as follows:
  - Disease data:
    - Disease data gives information about each disease and its constituent symptoms.
    - Internally, each symptom for a particular disease is rated as CRITICAL, IMPORTANT and OPTIONAL so as to give the expert system an idea about the most and least important symptoms of a particular disease.
    - For example, the disease data of *dengue* looks like this:

```
Dengue = {
    Fever : CRITICAL,
    body_pain : IMPORTANT,
    joint_pain : IMPORTANT,
    pain_behind_eyes : OPTIONAL,
    rash : OPTIONAL,
    body_pain_muscles : OPTIONAL,
    'name' : "dengue"
}
```

- Questions data:

- Questions data is the question that will be asked for each symptom in order to confirm the existence of the symptom in the patient.
- One question corresponds to one symptom. These questions are divided into top questions and specific questions.
- Top questions are for confirming if the patient shows signs of a particular symptom and specific questions are for asking about specific details of a particular symptom.
- Each question is tagged with the name of the symptom. The tags play an important role in deciding the hierarchy of the symptoms when they are loaded into various expert system data structure like the symptom\_validity\_table and scratch\_pad.
- A sample top question and its specific question would look like:

```
# Top question to confirm if body pain exists.
'body_pain':{
'question':"Do you have body pain?",
'response':['Yes','No'],
'response_type':'ruledchar',
}

# Specific question asking where exactly the
# body pain might be.
'body_pain_head':{
'question':"Does your Head hurt?",
'response':['Yes','No'],
'response_type':'ruledchar',
'serial':0,
}
```

#### 4.1.2 Dynamic data

- Dynamic data involves the data that is provided to the system by real human users as a result of the conversations that they have with the system. This data is stored in the Redis database as key-value pairs on a per user basis.
- There are primarily 4 data tables, or Redis ‘hashes’, where the data is stored.
  - **Per user basic data**
    - This is a Redis hash that is unique to each user.

- It is stored in form of chat\_id + “:basic\_data”. This hash contains basic data about the user, it contains the following attributes.
  - Age
  - Height
  - Weight
  - Gender
- **Per user symptom data**
  - This is a Redis hash that is again, unique to each user.
  - It is stored in the form chat\_id + “symptoms”. This hash contains keys that are symptoms and the corresponding values of each key corresponds to an integer that represents the number of times the user has reported that particular symptom till date.
  - The key is not created until the user reports that symptom. An example database looks like this:
    - Fever – 2
    - Body\_pain – 1
    - Fatigue – 1
- **Global symptom count**
  - Since we use machine learning algorithms for reporting symptoms, a hash called GLOBAL\_SYMPTOM\_COUNT keeps a track of the number of times ALL the symptoms have been answered till date.
  - It contains key-value pairs where the keys are symptoms and the values are all initially set to zero.
  - Whenever any user reports a particular symptom, the count of that symptom in this hash is incremented by 1.
- **Global username chat id store**
  - This is a simple key value store that stores the username with the corresponding chat ID.
  - A username-chat ID entry is made into this table as soon as a message arrives in the server.

- This table helps us keep a track of the number of users we have had and their corresponding chat IDs.
- A sample store with two users 'v0dro' and 'jaideepkekcre' will look something like this:
  - V0dro – 1234
  - Jaideepkekcre - 23456

## 4.2 GUI Design

We use the Telegram GUI. Our only addition to that provided by Telegram is custom keyboards. Hence, a snapshot of a conversation in progress would look something like this:



Fig 4.2 Telegram GUI.

## 4.3 Operational Details

### 4.3.1 Server

The server module is the primary gateway in which the system can converse with the outside world. It is basically an asynchronous, non-blocking, multi-threaded, multi-process python program that polls the

Telegram chat server for incoming messages, passes them to the core logic for actual processing and returns the result to the user.

We have used the python-telegram-api library as a useful Python wrapper that makes communicating with Telegram very simple. It allows for easily setting method hooks that are triggered whenever a particular action takes place on part of the Telegram chat bot. It also lets our program poll the Telegram server every 0.1 seconds with the start\_polling() method.

For example, to call accept\_message method whenever a message that is not a Telegram ‘command handler’, poll the Telegram server:

```
updater.addTelegramMessageHandler(accept_message)
# start polling the Telegram server...
updater.start_polling(0.1)
```

The accept\_message method internally uses a shared queue (an instance of multiprocessing.QUEUE) that is shared amongst all 3 processes that the server uses. It just passes the relevant data from the received messages inside the queue and exits, so that other incoming messages are not blocked as a result of processing/network lag experienced by any given message. Internally, the accept\_message method looks like this:

```
d = {
    # populate dict with useful information about message
}
MESSAGE_QUEUE.put(d)
```

Here, ‘MESSAGE\_QUEUE’ is the global shared queue.

The queue is continually polled for pending messages by a separate process that sends them for processing to the expert system. This is done by first sending it to the ‘dispatcher’ module. The code that does this can be briefly summarized as:

```
While True:
if messages_present_in_queue:
user_info = get_most_recent_message_from_queue
    m = send_message_to_dispatcher_for_processing(user_info)

add_machine_response_to_dispatch_queue(m, user_info)
```

As you can see, as soon as the process detects that a message is present in the queue, it is immediately sent to the dispatcher for processing, and the returned message is added to the dispatch queue, which is a



queue shared with another process whose sole job is to send messages back to the user. Thus message processing happens independent of network lags that might be experienced while sending messages back to users.

The third process is a multi-threaded process whose sole job is to poll the shared dispatch queue for messages, and send them back to the concerned Telegram user. Each message in the queue is sent over a different thread so that network experienced for a single message does not affect the messages present in the queue.

### 4.3.2 Dispatcher

The dispatcher is the module that sits right behind the server to accept and process messages from users. A single dispatcher instance is created by the server and the messages are then sent to that instance by calling the `dispatcher.run_dispatcher()` method.

Internally, the dispatcher maintains a Python dictionary of the currently active conversations that it is having. This dictionary is called `object_list`. The keys of this dict are the chat IDs of users that are unique to each user, and the values are objects of type `core`. The `core` class is the one that is actually responsible for processing of messages.

Whenever the dispatcher receives a new message, it searches the `object_list` if the chat ID of the incoming message is already present. If yes, it is interpreted as receiving a reply for a message that was previously sent to the user, and the message is directly sent into the `core` object that the chat ID is associated with. If no, a new entry is created in the `object_list` dictionary by associating the chat ID of the message with a newly manufactured `core` object.

As we will see soon, the `core` class is responsible for keeping track of a conversation, which it does by keeping some variables that define the current state of the conversation. The beauty of this approach is that it becomes very simple to keep track of a conversation simply by persisting the `core` objects in memory until the conversation is over.

When a conversation is over, the `core` object associated with the chat ID is removed from the dict, and thus eventually de-allocated from memory by the Python Garbage Collector.

Roughly, the most important function of the dispatcher, as summarized above, can be represented by the following code snippet:

```
object_list=dict()
if conversation_with_chat_id_over(chat_id):
    object_list[chat_id]=assign_new_core_object()

core_object=object_list[chat_id]
core_object.process_message_and_return_reply()
```

### 4.3.3 Core

The core module is called by the dispatcher. One instance of the core class will exist for each independent conversation that is in progress. Core contains variables and methods that let it keep track of the conversation from a top level perspective, i.e. it can only know if a conversation is in progress or is over. Intermediate states are not the concern of core.

It instantiates the expert system and manages sending and receiving data from it, which it eventually passes back to the dispatcher class for sending back to the user. All pre-processing on incoming messages like spell checking, removal of punctuation, etc. are done by core before the message is even passed to the expert system. It returns messages in the form of a Python dict and if the conversation is over, returns 'None'.

The core logic that makes all this happen can be summarized as:

```
Expert = expert_system()# instantiate expert system module

if expert.done == 1:# conversation is over
    return None
else:# conversation still in progress
    return_message = expert.run_expert_system(params_from_incoming_message)
    return return_message
```

### 4.3.4 Question and Response data

For the expert system to work as expected, it is essential to have a knowledge store. The knowledge for our chat bot is represented by python files kept in the data/ folder. Each question has certain attributes that tell more information about the question, like the response that is supposed to show up on the Telegram custom keyboard once the question is presented to the user. The different attributes of a question can be:

1. **Question string** – This is the actual question that will be sent across to the user.

2. **Response** – This is a list of responses that will be shown to the user that denote the answers that he/she can send to the system via the Telegram custom keyboard.
3. **Serial** – This is an internal integer that denotes the order in which the question should be asked.

There are two types of questions that can be asked to a user:

- Basic data questions:
  - Basic data questions are asked for collecting basic personal data of each patient like age, gender, weight and height.
  - These are questions that must be asked when the patient first begins a conversation with the system.
  - The questions are kept as a Python dictionary in the basic\_data.py file.
  - For example, a question that asks a user their age looks like this:

```
'age':{
  'question':"What is your age?",
  'response':['0-15','15-25','25-40','40-50','>50'],
  'response_type':'ruledchar',
  'serial':1
}
```

- Symptoms related questions:
  - Symptom related questions are the questions that will be asked to patients for confirming whether they are showing signs of a certain symptom or not.
  - They are able to gather either yes/no style responses or specific responses based on pre-decided parameters.
  - These questions are further divided into two types:
    - **Top questions** – A top question is a question that confirms if a particular symptom is present or not.
    - **Linked questions** – Linked questions are questions that are used for asking for more information about a particular symptom in case the patient replies affirmatively to the top question of that symptom.
  - The top questions are loaded into a file called top\_questions.py. Each of these questions is designed such that they give an affirmative response of the symptom being present or

absent. For example, a top questions that would confirm the presence of body pain looks like this:

```
'body_pain':{
  'question':"Do you have body pain?",
  'response':['Yes','No'],
  'response_type':'ruledchar',
}
```

- Since the linked questions are about gathering more information about a particular symptom, they placed in separate files, each of which is named after the symptom that the questions in the file are about. So questions that get more information of body pain are placed in a file called `body_pain.py`. A sample linked question that asks whether the body pain a patient is facing is in the head would look like so:

```
'body_pain_head':{
  'question':"Does your Head hurt?",
  'response':['Yes','No'],
  'response_type':'ruledchar',
  'serial':0,
}
```

#### 4.3.5 Expert System

This is the module that contains the actual expert system for inferring the meaning and context of replies. The decision of the next question takes place in `DoctorSkyNet` module. The expert system decides the state of the conversation by keeping track of it with the status variable. Most of the core logic of the expert system lies inside the `run_expert()` method.

If the status of the expert system is *1*, it implies that the expert system is in a state where the first message from the user has just arrived and he/she hasn't actually replied to any question posted by the system. This state is necessary because it determines if the response is sent to the `DoctorSkyNet` module. Once the first reply is delivered to the user, the expert system transitions to state *2*.

When in state *2*, the expert system is expecting a reply from the user, which it will send to `DoctorSkyNet` for further processing. If `DoctorSkyNet` returns a `None`, the expert system assumes that the conversation is over and goes to *3*, which implies that a message stating that the test is over should be sent to the user.

An overall functioning of the expert system can be denoted like this:

```

if user_has_begun_conversation():
    status=1

if status == 1:# First question is being asked.
    Status = 2# First question asked. Now expecting user to reply.
    get_first_question_from_doctor_skynet()

if status == 2:# System expecting a reply from user.
    send_user_response_to_doctor_skynet_and_get_next_question()
    if no_more_questions_remain():
        state=3# State 3 represents that the conversation should be terminated.
        done=1# Indicates that the conversation is over.
        return_conversation_done_to_core()

    return_question_to_core()

```

#### 4.3.6 Doctor Sky Net

This module is where the real magic happens. Its primary purpose is to use intelligent algorithms to read the response from the user, keep a track of the symptoms that the user has already answered about, and to serve questions to the user that are most relevant to him based on a data base of diseases that is maintained.

Each disease is represented as a ‘bucket’ (using the ‘Buckets’ module as we’ll see later). Each bucket is associated with certain symptoms. The algorithms employed by DoctorSkyNet read the state of these buckets and serve the most relevant question to the user. This helps us narrow down the number of questions that need to be asked by the system in order to reach a possible diagnosis.

The algorithms that are used by DoctorSkyNet can be summarized as follows:

- Algorithm minus one
  - Role: Clustered symptom identifier.
  - Finds the symptom that ALL the patients are suffering from.
  - Uses Redis.
  - Does not interact with buckets.
  - Steps:
    1. Get all symptom, ‘Yes’ count from Redis via ORM.
    2. Find symptom with highest Yes count.

3. Return that symptom.

- Algorithm zero

- Role : Most probable symptom finder , patient history based
- Finds the symptom that the patients is PRONE to.
- Uses Redis.
- Does not interact with buckets.
- Steps:
  1. Get Patient's historical symptom, yes count from Redis via ORM.
  2. Find symptom with highest Yes count.
  3. Return that symptom.

- Algorithm one

- Role: Find symptom with highest score across buckets.
- Finds the symptom that will fill largest number of buckets.
- Does not interact directly with buckets.
- Steps:
  1. Get symptom score dict.
  2. Find symptom with highest score.
  3. Return that symptom.
  4. Update buckets, remove symptom from all score dicts.

- Algorithm two

- Role : Find Critical symptom with highest score across buckets
- Finds the symptom that will remove largest number of buckets if answered in negative.
- Does not interact with bucket directly
- Steps:
  1. Get Critical symptom score dict.
  2. Find symptom with highest score.
  3. Return that symptom.
  4. Update buckets, remove symptom from all score dicts.

- Algorithm three

- Role: Find symptom with highest cumulative remaining score in each bucket.
- Finds the symptom that will remove largest number of buckets if answered in negative.

- Steps:
  1. Get Bucket.
  2. Find symptom with highest score in Bucket.
  3. Record that symptom, score in data structure.
  4. If buckets left go to step 1.
  5. Find symptom with highest score in data structure
  6. Return that symptom

The overall functioning of the module is controlled by 2 stages. Each stage denotes what part of the conversation the system is in currently. The first stage does a primary check if the basic data of the patient has been acquired. It does this by checking in the redis database for the basic data. If not, it initiates a conversation with the patient to gain more insights about their basic personal data like age, gender, weight and height.

Once the basic data is in place, DoctorSkyNet moves to the second stage and proceeds to query the patient for symptoms based on the above algorithms. It sets the 'done' variable to 1 once sufficient data about symptoms has been acquired by the system.

The overall working of DoctorSkyNet can be summarized as follows:

```
While conversation_in_progress():
    update_bucket_fractions()
if basic_questions_unanswered():
    init_basic_questions_conversation()

if symptom_queries_not_over():
    init_symptom_query_conversation()
```

#### 4.3.6 Question Interface:

The question interface class stores the details of each question and makes it quickly accessible to any other class needing access to questions in a relatable format. It has the same attributes as any question dictionary in the data store.

#### 4.3.7 Disease Buckets

The disease buckets are used for representing diseases. Each disease is assigned its own bucket and each bucket consists of multiple symptoms.

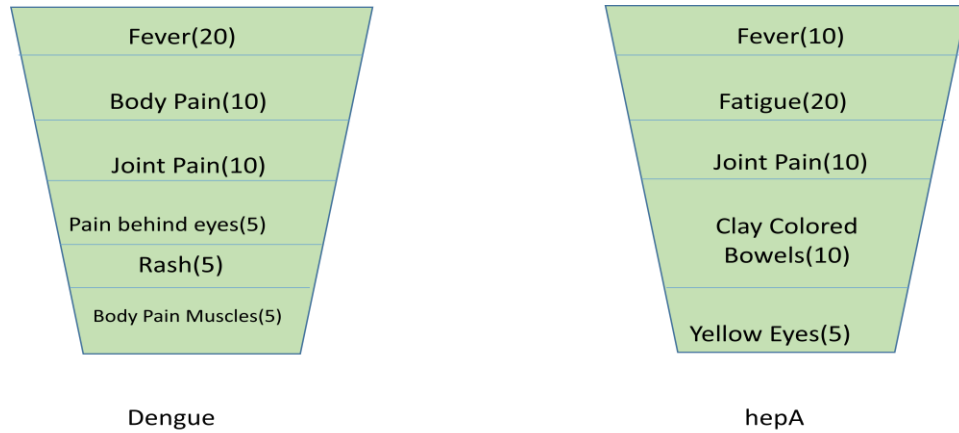


Fig 4.3 Disease Buckets

#### 4.3.8 Redis connection

This module maintains a connection with the redis database by using the py-redis library as a useful Python wrapper. It exposes a helpful pythonic interface to the redis database.



## Chapter 5

### Testing

#### 5.1 Acceptance Testing

Acceptance testing is a test conducted to determine if the requirements of a specification or contract are met. For our acceptance tests, we have mainly conducted tests in two forms – manual testing of the chat bot with real human users, and checking various internal data structures and schemas for consistency with the tests.

#### 5.2 Unit Testing (module wise testing)

Python allows for a helpful feature that lets you add testing code for a module at the bottom of the file. Encapsulating unit tests inside an `if __name__ == '__main__':` block runs only that code when the entire file is executed on the command line by passing it as an argument to the python interpreter.

Thus, unit tests per module are placed at the bottom of the file that contains a module, and the tests are written there itself. This also serves to unite code and tests and makes it easier to change behaviour if need be. As an example, the unit tests for the **DoctorSkyNet** class look this:

```
if __name__ == '__main__':
    obj=DoctorSkyNet()
    obj.askdoctor()
    obj.askdoctor("Yes, High (> 103 F)")

    obj.askdoctor("No")

    obj.askdoctor("Yes")
    obj.askdoctor("No")
    obj.askdoctor("No")
    obj.askdoctor("Yes")
    obj.askdoctor("Yes")
    obj.askdoctor("No")
    obj.askdoctor("Yes")
    obj.askdoctor("Yes")
    obj.askdoctor("Yes")
    obj.askdoctor("Yes")
    obj.askdoctor("Yes")
    obj.update_fractions()
    printobj.fraction
    print"***"
```

### 5.3 Integration Testing

Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing code, in our case, is mostly combined with the unit testing code at the bottom of files so that if a module is dependent on another module, the interfaces can be defined then and there without much hassle or scaffolding code.

In case many modules need to be combined or if we feel that a separate test harness is required for building integration tests, we make use of the python **unittest** module for writing test. Unittest exposes a very friendly and intuitive API for writing integration tests by way of combining them inside class and inheriting from the **unittest.TestCase**. For example, the tests that validate the combination of the question interface module with database look like this:

```

Class TestDBWithScratchPad(unittest.TestCase):
Def setUp(self):
From scratch_pad import scratch_pad
From db_store import db

# setup necessary data structures and types

Def test_get_next_unanswered_question(self):
# test getting next unanswered question

Def test_get_specific_question(self):
# test getting specific question

```

## Chapter 6

## Results and Discussion

## 6.1 Main GUI Snapshots

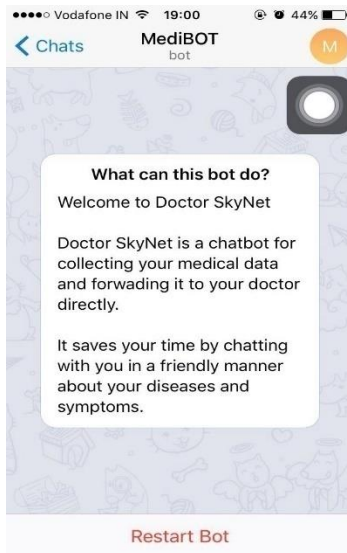


Fig 6.1 (a) GUI 1



Fig 6.1 (b) GUI 2



Fig 6.1 (c) GUI3



Fig 6.1 (d) GUI 4



Fig 6.1 (e) GUI 5



Fig 6.1 (f) GUI 6



Fig 6.1 (g) GUI 7



Fig 6.1 (h) GUI 8

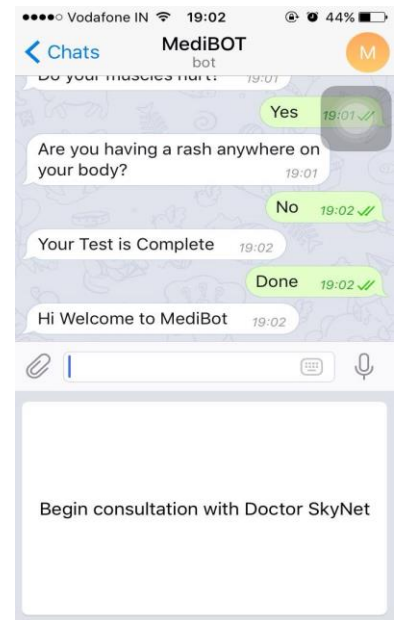


Fig 6.1 (i) GUI 9

The above diagrams are snapshots of the Telegram iPhone app at different stages of evaluation. They try to show a full conversation that the user can have with the system. Notice the changing keyboards and responses in each snapshot, it should give you a good idea of what the actual experience is like.

## 6.2 Terminal snapshots

In the terminal snapshot below you can see the output that is displayed by the server when it receives a message over telegram from the user. The red line states that the user's object was not already there in memory and that it is being created, meaning that a new conversation with a user has started and the message received is not sent as a reply to a previous response message. The logger does not create new objects when another response is received from the user, which goes to show that the same object stores the user's state as long as the conversation is in progress.

The figures 6.2 a-c show this behaviour in detail. You can see the conversation start in (a), then progress in (b) and then finally end in (c).

```

14012:W 17 Mar 18:58:36.963 # Creating Server TCP listening socket *:6379: bind: Address already in use
User with chat id90417424 not found , creating new object
USER OBJECT CREATED WITH CHAT ID: 90417424

hepA : LOADED
dengue : LOADED
User Response is Begin consultation with Doctor SkyNet
Begin consultation with Doctor SkyNet
0.0
using algo-1-
Do you have a fever?
['Yes, High (> 103 F)', 'Yes, Mild (101-103 F)', 'Yes, Very Mild (99 - 101 F)', 'No']
\input is :Begin consultation with Doctor SkyNet
response is :['Do you have a fever?']

User Response is Yes, High (> 103 F)
Yes, High (> 103 F)
0.181818181818
using algo-1-
Are you having joint pain?
['Yes', 'No']
options are:['Yes', 'No']
\input is :Yes, High (> 103 F)
response is :['Are you having joint pain?']

User Response is Yes
Yes
0.363636363636
Using algo-2-
Are you experiencing any fatigue?
['Yes', 'No']
options are:['Yes', 'No']
\input is :Yes

```

Fig 6.2 (a) Server Snapshot 1

```

Activities | Terminal | Thu 19:02
./run.sh

./run.sh x ipython x pry x
\ninput is :Yes, High (> 103 F)
response is :['Are you having joint pain?']

User Response is Yes
Yes
0.363636363636
Using algo-2-
Are you experiencing any fatigue?
['Yes', 'No']
options are:['Yes', 'No']
\ninput is :Yes
response is :['Are you experiencing any fatigue?']

User Response is No
No
hepA removed
0.545454545455
Using algo-2-
None in algo two
using algo-3-
Do you have body pain?
['Yes', 'No']
options are:['Yes', 'No']
\ninput is :No
response is :['Do you have body pain?']

User Response is Yes
Yes
0.727272727273
using algo-3-
Are you experiencing pain behind the eyes?
['Yes', 'No']
options are:['Yes', 'No']

```

Fig 6.2 (b) Server Snapshot 2

```

Activities | Terminal | Thu 19:02
./run.sh

./run.sh x ipython x pry x
0.727272727273
using algo-3-
Do your muscles hurt?
['Yes', 'No']
options are:['Yes', 'No']
\ninput is :No
response is :['Do your muscles hurt?']

User Response is Yes
Yes
0.818181818182
using algo-3-
Are you having a rash anywhere on your body?
['Yes', 'No']
options are:['Yes', 'No']
\ninput is :Yes
response is :['Are you having a rash anywhere on your body?']

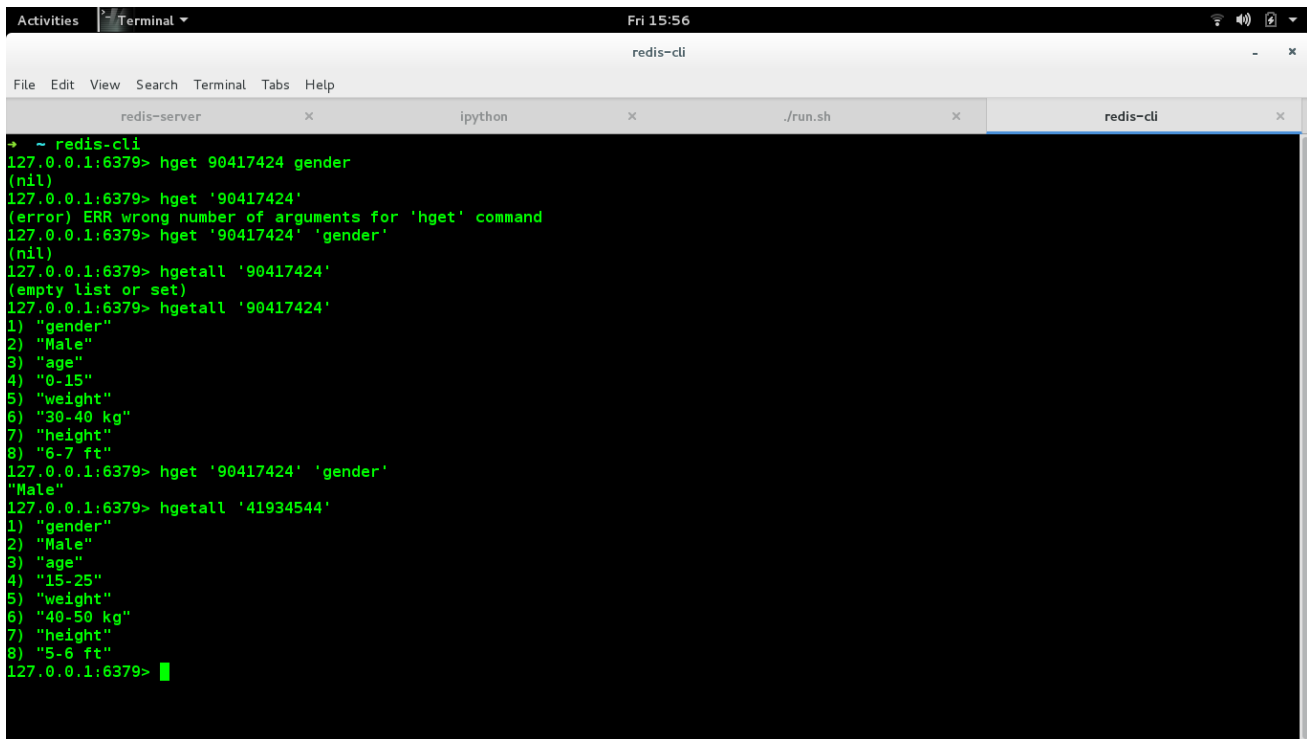
User Response is No
No
0.818181818182
using algo-3-
None in algo three
Done
All Questions done!
\ninput is :No
response is :['Your Test is Complete']

Chat ID : 90417424 removed
\ninput is :Done
response is :['Hi Welcome to MediBot']

```

Fig 6.2 (c) Server Snapshot 3

The figure below is the Redis database dump for the conversation that was showed in Fig. 6.2 (a) to Fig. 6.2 (c). It shows the values that were stored in the database as per the user ID.



```
Activities Terminal Fri 15:56
redis-cli
File Edit View Search Terminal Tabs Help
redis-server x ipython x ./run.sh x redis-cli x
→ ~ redis-cli
127.0.0.1:6379> hget 90417424 gender
(nil)
127.0.0.1:6379> hget '90417424'
(error) ERR wrong number of arguments for 'hget' command
127.0.0.1:6379> hget '90417424' 'gender'
(nil)
127.0.0.1:6379> hgetall '90417424'
(empty list or set)
127.0.0.1:6379> hgetall '90417424'
1) "gender"
2) "Male"
3) "age"
4) "0-15"
5) "weight"
6) "30-40 kg"
7) "height"
8) "6-7 ft"
127.0.0.1:6379> hget '90417424' 'gender'
"Male"
127.0.0.1:6379> hgetall '41934544'
1) "gender"
2) "Male"
3) "age"
4) "15-25"
5) "weight"
6) "40-50 kg"
7) "height"
8) "5-6 ft"
127.0.0.1:6379> █
```

Fig 6.3 Redis Database Dump

## **Chapter 7**

### **Conclusion**

---

After studying various expert systems, chat bots and getting feedback from people involved in the medical profession, it can be concluded that:

- A chat bot based medical expert system can be very useful for reducing the stress that doctors have to go through on a daily basis.
- However, the results of the system should still be verified by a legitimate doctor before they are furnished to the patient.
- Such a system can be perfected to incorporate all sorts of diseases.



## References

---

- [1] Hallili, Amine. *"Toward an ontology-based chatbot endowed with natural language processing and generation."* 26th European Summer School in Logic, Language & Information. 2014.
- [2] Kohane, Isaac Samuel. *Temporal reasoning in medical expert systems*. Boston Univ., MA (USA), 1987.
- [3] Shortliffe, Edward H., and Lawrence M. Fagan. *"Expert systems research: modeling the medical decision making process."* (1982).
- [4] Hill, Jennifer, W. Randolph Ford, and Ingrid G. Farreras. *"Real conversations with artificial intelligence: A comparison between human–human online conversations and human–chatbot conversations."* *Computers in Human Behavior* 49 (2015): 245-250.
- [5] Dutta, Soumitra. "Temporal reasoning in medical expert systems." *Engineering of Computer-Based Medical Systems, 1988., Proceedings of the Symposium on the*. IEEE, 1988.
- [6] <http://www.wikipedia.org/>
- [7] <http://www.obofoundry.org/>
- [8] <https://github.com/python-telegram-bot/python-telegram-bot>
- [9] <https://docs.python.org/3/>

## Appendix A

---

### A.1 ACMKeywords

1. J. Computer Applications
  1. J.3 LIFE AND MEDICAL SCIENCES
    - i. Health
    - ii. Medical Information Systems
2. H. Information Systems
  1. H.1.2 User/Machine Systems
    - i. Human factors
    - ii. Human information processing
3. I. Computing Methodologies
  1. I.2 ARTIFICIAL INTELLIGENCE
    - i. I.2.1 Applications and Expert Systems
      - A. Medicine and science.
      - B. Natural language interfaces.

### A.2 IDEA Matrix

Increase	Drive	Educate	Accelerate
Improve	Deliver	Evaluate	Associate
Ignore	Decrease	Eliminate	Avoid

#### 1. I

- **Increase** The productivity of doctors and other medical staff involved in medical history taking.
- **Improve** The interaction of the first meeting between the doctor and patient.
- **Ignore** Non-intelligent solutions to problems.

#### 2. D

- **Drive** Advancement of chat bot based medical expert systems.
- **Deliver** A chat bot based medical expert that is aware of a certain set of diseases.
- **Decrease** The time spent by doctors and medical staff in taking history of patients.

#### 3. E

- **Educate** New students of AI/ knowledge based system to develop better systems.
- **Evaluate** The best algorithms for creating accurate medical expert systems.
- **Eliminate** The history taking phase of patients.

#### 4. A

- **Accelerate** The speed of check-ups.
- **Associate** Advanced technology with medical science.

- **Avoid** Wasting time on tasks that can be performed by machines.

### A.3 Mathematical Model

#### 1. System for Patient

$$S = \{S_P, S_D, S_A\}$$

$S_P$  = System for Patient

$S_D$  = System for Doctor

$S_A$  = System for Admin

#### 2. System for Patient

$$S_P = \{I, O_P, F_P, Er\}$$

$I$  = Input

$O$  = Output

$F$  = Functions

$Er$  = Error States

$$I = \{s_1, s_2, s_3 \dots s_n\}$$

$S$  = Sentences

$$S = \{W_1, W_2, W_3 \dots W_n\}$$

$W$  = Words

$O_P$  = **Prognosis**

$$F_P = \{FL, FUR, FKE, FNLP, FDB\}$$

$F_L$  = Login Method

$F_{UR}$  = User Registration

$F_{KE}$  = Keyword Extraction

$F_{NLP}$  = NLP Method

$F_{DB}$  = Database Method

$$\text{SessionID} = F_L(W_n)$$

**Registration =  $F_{UR} (W_n)$**

**Tokenized Words =  $F_{KE} (S_n)$**

**Prognosis =  $F_{NLP} (Tokens)$**

**Storage Activity =  $F_{DB} (Data)$**

**Er = { FAILLOGIN, FAIL<sub>NLP</sub> }**

### 3. System for Doctor

**S<sub>D</sub> = { I, O<sub>D</sub>, F<sub>D</sub>, Er }**

I = Input

O = Output

F = Functions

Er = Error States

**I = { O<sub>P</sub>, PID }**

PID = Patient ID (text)

**O<sub>D</sub> = VerifiedPrognosis**

**F<sub>D</sub> = { FDR, FDV, FLogin, F<sub>DB</sub> }**

F<sub>DR</sub> = Doctor Registration

F<sub>DV</sub> = Verification of Prognosis

F<sub>Login</sub> = Doctor Login

F<sub>DB</sub> = Database Method

**SessionID =  $F_{LOGIN} (W_n)$**

**Doctor Registration =  $F_{DR} (W_n)$**

**Verified Prognosis =  $F_{DV} (S_p)$**

**Storage Activity =  $F_{DB} (Data)$**

### 4. System for Admin

**Er = { FAIL<sub>VERIFICATION</sub> }**

**S<sub>A</sub> = { I, O<sub>A</sub>, F<sub>A</sub>, Er }**

I = Input

O = Output

F = Functions

Er = Error States

**I = {W<sub>1</sub> W<sub>2</sub>... W<sub>n</sub>, Rules}**

Rules = Inference rules for business logic

**O = Working System**

**F<sub>A</sub> = {F<sub>R</sub>, F<sub>Login</sub>, F<sub>DB</sub>, F<sub>RE</sub>}**

F<sub>DR</sub>=Registration

F<sub>RE</sub>= Rules Input

F<sub>Login</sub>= Doctor Login

F<sub>DB</sub> = Database Method

**SessionID = F<sub>LOGIN</sub> (W<sub>n</sub>)**

**Registration = F<sub>DR</sub>(W<sub>n</sub>)**

**Verified Knowledge = F<sub>RE</sub>(W<sub>n</sub>)**

**Storage Activity = F<sub>DB</sub>(Data)**

**Er = {FAIL<sub>Rule Input</sub>}**

## Appendix B

---

### B.1 Individual Contributions

- Jaideep Kekre
  - Planning: Figuring out the actual components that will go into building the system and coming up with a concrete top level architecture that will be extensible and scalable. Quantizing and documenting ideas in such a way that they can be easily implemented.
  - Documentation: Documenting the team's efforts throughout the course of the project, coordinating with concerned faculty and providing them with timely updates. Creating Mathematical Models, conducting feasibility studies, etc.
  - Doctor Sky Net programming: Program and deploy the core algorithms for the expert system.
- Sameer Deshmukh
  - Server Side programming: Program and deploy the server side code. This includes the Business logic and Database interface modules and other peripheral functionality to make them work with the rest of the system.
  - Technology: Finalizing the technology to be used after having received the design and architectural specifications. Finalizing automated test procedures.
- Bishal Kumar
  - System Interface: Responsible for designing and programming user interfaces, also creating meaningful connections between said UI and the relevant backend.
  - Design and Aesthetics: Decides on the design principles of the whole project including presentations, reports and any UIs.
  - UML diagrams: Responsible for creating the UML and other design diagrams.
- Sandeep Gaikwad
  - Testing: Decide on a test plan and execute non-automated tests.
  - Documentation: Work with Jaideep on the documentation

(Signature of Guide)

Prof. C. A. Laulkar

# DESCRIPTION OF A CHATBOT BASED MEDICAL EXPERT SYSTEM

Sameer Deshmukh, Jaideep Kekre ,Dept. Of Computer Engineering, Sinhgad College Of Engineering

**Abstract:** Doctors in India have to work long hours and see hundreds of patients a day. A lot of time is spent by doctors for gathering the personal and medical history of patients with respect to their demography and present and past symptoms. Our project enhances the first doctor-patient interaction where the doctor asks the patient certain questions for noting down their history

**Index terms:** Expert system, Medical decision modelling, Chatbot.

## I .Introduction

The aim of this project to develop a chat bot based medical expert system. We believe that a patient's basic history taking can be done by having a conversation with a friendly chat bot, so that the doctor is freed from this activity and precious time is saved. This project will only aim to act as a middle man between the patient and doctor. It works by letting the patient chat with the system so that he/she can report their personal and medical history to the computer, which the computer keeps a track of. The system will then try to infer a diagnosis from this interaction and report it directly to the doctor by sending the doctor an email.

Expert system is a computer system that emulates the decision-making ability of a human expert. As described above, most doctors in India are over-burdened by the number of patients that they have to see on a daily basis. We have come up with a novel

solution to this problem by creating a chatbot based medical expert system that can have a friendly conversation with a patient for making note of their medical history. For this purpose we have used the Telegram messaging app as a front-end for the patient. Telegram is a widely used, cross-platform, open source and highly secure chat application that lets bots have conversations with users. It does this through a RESTful API which any chat bot can interface with.

What's in the scope?

1. Let's patients chat with the system to extract an accurate history from them.
2. Let's doctors see the medical history of their patients through a similar chat interface.
3. Tries to zero in on the disease that the patient might have by relating symptoms to each other.

And what's out of scope:

1. Does not aim to replace doctors.
2. Cannot detect and accurately extract symptoms of complex diseases like cancer or diseases where manual check-up or tests are essential.
3. Limited by the knowledge base.

**There will primarily be 3 classes of users:**

### **Patients**

The primary users of the system. Patients will chat with the system through a chat interface. They will communicate their symptoms to the system and the system will ask them relevant questions to figure out a diagnosis. They will chat with the system and engage in a chat for determining the exact disease. Patients are the most important users of the system.

### **Doctors**

The doctors will use the system for accessing data about patients. They will be able to view all the patients whose data they have access to and also view specific details about each patient.

### **Admin**

The admin will be responsible for maintenance and upkeep of the system. The admin will be sole person who will be able to set permissions for doctors who would want to access data of a given patient.

## **II. Operating Environment and Interfaces**

Operating environment will be as follows:

- Any UNIX based operating system. We will be using Debian/Ubuntu.
- Internet connection.
- Python interpreter.
- Various python libraries like python-telegram-api, threading, multiprocessing, etc.
- Redis.
- Computer running commodity hardware.

## **Design and Implementation Constraints**

A major concern is maintaining the privacy of user data. Only specific doctors should be able to access the symptoms related data of certain patients and that too with their consent. Admin access to this data will be restricted to the chief maintainers of the system. Thus data will have to be stored in a very secure manner and it cannot be distributed without specific legal permission from patients. The front end of the system (user facing) will depend on the Telegram mobile app. We will need to conform to the standards set by Telegram for chat communication between client and server. All the server side code will be written in Python and the database will be Redis. We will primarily use HTTP get and post requests to send and receive data over the network.

### **A. Interface between patient and system:**

This interface is already created by Telegram, and we will be using the same to take information from the patient and deliver the relevant information back to them. It predominantly features a text box for input of text, a chat window that displays previous conversations, a keyboard for typing responses and a 'send' button for sending this information to the system.

### **Interface between doctor and system:**

The only information that the doctor expects from the system is information about his/her patients. This will be delivered to them via their email account.



### Interface between admin and system:

The admin takes care of assigning doctors to patients. This will be done through a command line interface.

### Patient chats with the system:

**Stimulus:** Patient sends a text message describing his ailment.

**Response:** Chat bot pulls the users details from the database into active memory for further processing. System will initiate a chat and try to diagnose the patient's disease. This will involve read/write calls to the data base and business logic modules. Once the chat reaches a 'finished' state, an appropriate message will be sent to the user indicating this.

**Stimulus:** Patient replies with the Telegram in-app keyboard to the question asked by the system.

**Response:** The chat bot records this response, and after processing is done by the business logic module, an appropriate reply is send back to the user. This might be another question to prompt the patient for more information or a message announcing that the conversation is over.

### B. Doctor-system interaction:

**Stimulus:** A patient whose data has been authorised to be viewed by a particular doctor has just successfully completed a conversation with the system.

**Response:** The system will fetch the relevant patient details about the patient and return it to the doctor in a readable and understandable format. The information will be sent to the doctor as an email. The responses will typically concern obtaining more information about a particular patient.

### C. Admin-system interaction:

**Stimulus:** Admin wants to authorize a doctor to be able to receive information about a given patient.

**Response:** System associates doctor credentials with the patient and authorises doctor to access patient information.

## III. System Modules and Architecture

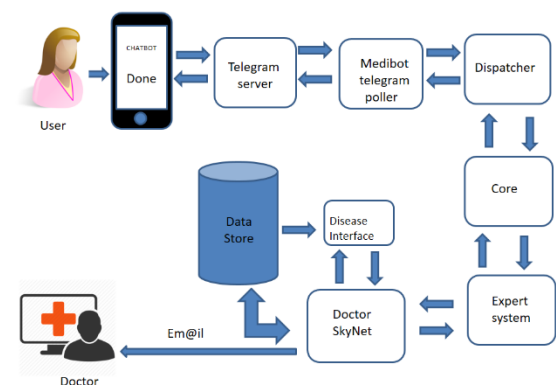


Figure 1. Architecture Diagram of the System

### A .Server module

This module is responsible for receiving and sending messages by interacting with the Telegram server. Additionally it is also responsible for dispatching messages to the expert system modules and getting replies from them.

### B. Database communication module

This is the module that is completely responsible for read/write calls to the database. The data base module will supply the relevant user data to the chat bot and receive data to be stored in the data base

### C. Dispatcher module

The dispatcher acts like a router. It allocates a single object to an independent conversation.

### D. Question and response data store

This is a static data store which stores the data about questions and responses.

### E. Core module

This module is a state aware module that is spawned on a per conversation basis. It is aware of the state that the conversation is in at any point of time.

### F. Expert System module

The Expert System module sits between the core and Doctor Sky Net modules. It takes care of handling conversation states and communication between Core and Doctor SkyNet.

### G. Doctor SkyNet module

This is the core module that contains various crucial algorithms for actual functioning of the entire system.

## IV. Database schema

The system as a whole relies on two kinds of data – hard coded data and dynamic data. They are represented as follows:

#### A .Hard coded data

This data either pertains to symptoms, diseases or questions. It highly important data that must be in a particular format. It is the primary knowledge store of the expert system.

It is stored as Python native data structures like dictionaries, lists, strings and numbers – all in dedicated python files. The reason for keeping these hard coded and not in any external database is to completely nullify the possibility of unwanted elements making changes to the data, which might prove disastrous for the system as whole.

The two major types of hard coded data are as follows:

#### B .Disease data:

Disease data gives information about each disease and its constituent symptoms.

Internally, each symptom for a particular disease is rated as CRITICAL, IMPORTANT and OPTIONAL so as to give the expert system an idea about the most and least important symptoms of a particular disease.

For example, the disease data of *dengue* looks like this:

```
dengue = {  
    fever: CRITICAL,  
    body_pain: IMPORTANT,  
    joint_pain: IMPORTANT,  
    pain_behind_eyes: OPTIONAL,  
    rash: OPTIONAL,  
    body_pain_muscles: OPTIONAL,  
    'name': "dengue"  
}
```

#### C. Questions data:

Questions data is the question that will be asked for each symptom in order to confirm the existence of the symptom in the patient.

One question corresponds to one symptom. These questions are divided into top questions and specific questions.

Top questions are for confirming if the patient shows signs of a particular symptom and specific questions are for asking about specific details of a particular symptom. Each question is tagged with the name of the symptom. The tags play an important role in deciding the hierarchy of the symptoms when they are loaded into various expert system data structure like the symptom\_validity\_table and scratch\_pad.

A sample top question and its specific question would look like:

**# Top question to confirm if body pain exists.**

```
'body_pain':{
'question':"Do you have body pain?",
'response':['Yes','No'],
'response_type':'ruledchar',
}
```

**# Specific question asking where exactly the  
# body pain might be.**

```
'body_pain_head':{
'question':"Does your Head hurt?",
'response':['Yes','No'],
'response_type':'ruledchar',
'serial':0,
}
```

### C. Dynamic data

Dynamic data involves the data that is provided to the system by real human users as a result of the conversations that they have with the system. This data is stored in the Redis database as key-value pairs on a per user basis.

There are primarily 4 data tables, or Redis 'hashes', where the data is stored.

### D. Per user basic data

This is a Redis hash that is unique to each user.

It is stored in form of chat\_id + ":basic\_data". This hash contains basic data about the user, it contains the following attributes:

Age , Height, Weight, Gender

### E. Per user symptom data

This is a Redis hash that is again, unique to each user. It is stored in the form chat\_id + "symptoms". This hash contains keys that are symptoms and the corresponding values of each key corresponds to an integer that represents the number of times the user has reported that particular symptom till date. The key is not created until the user reports that symptom. An example database looks like this:

Fever – 2

Body\_pain – 1

Fatigue – 1

### F. Global symptom count

Since we use machine learning algorithms for reporting symptoms, a hash called GLOBAL\_SYMPTOM\_COUNT keeps a track of the number of times ALL the symptoms have been answered till date. It contains key-value pairs where the keys are symptoms and the values are all initially set to zero. Whenever any user reports a particular symptom, the count of that symptom in this hash is incremented by 1.

### G. Global username chat id store

This is a simple key value store that stores the username with the corresponding chat ID. A username-chat ID entry is made into this table as soon as a message arrives in the server. This table helps us keep a track of the number of users we have had and their corresponding chat IDs.

A sample store with two users 'v0dro' and 'jaideepkekke' will look something like this:

V0dro – 1234

Jaideepkekke - 23456

## V.Doctor SkyNet

This module is where the real magic happens. Its primary purpose is to use intelligent algorithms to read the response from the user, keep a track of the symptoms that the user has already answered about, and to serve questions to the user that are most relevant to him based on a data base of diseases that is maintained. Each disease is represented as a 'bucket' (using the 'Buckets' module as we'll see later). Each bucket is associated with certain symptoms. The algorithms employed by DoctorSkyNet read the state of these buckets and serve the most relevant question to the user. This helps us narrow down the number of questions that need to be asked by the system in order to reach a possible diagnosis.

The algorithms that are used by DoctorSkyNet can be summarized as follows:

### A. Algorithm minus one

- Role: Clustered symptom identifier.
- Finds the symptom that ALL the patients are suffering from.
- Uses Redis.
- Does not interact with buckets.

Steps:

- 2 Get all symptom, 'Yes' count from Redis via ORM.
- 3 Find symptom with highest Yes count.
- 4 Return that symptom.

### B.Algorithm zero

- Role : Most probable symptom finder , patient history based
- Finds the symptom that the patients is PRONE to.
- Uses Redis.
- Does not interact with buckets.

Steps:

1. Get Patient's historical symptom, Yes count from Redis via ORM.
2. Find symptom with highest Yes count.
3. Return that symptom.

### C. Algorithm one

- Role: Find symptom with highest score across buckets.
- Finds the symptom that will fill largest number of buckets.
- Does not interact directly with buckets.

Steps:

1. Get symptom score dict.
2. Find symptom with highest score.
3. Return that symptom.
4. Update buckets, remove symptom from all score dicts.

### D. Algorithm two

- Role : Find Critical symptom with highest score across buckets
- Finds the symptom that will remove largest number of buckets if answered in negative.
- Does not interact with bucket directly

Steps:

1. Get Critical symptom score dict.
2. Find symptom with highest score.
3. Return that symptom.
4. Update buckets, remove symptom from all score dicts.

### E . Algorithm three

Role: Find symptom with highest cumulative remaining score in each bucket.

Finds the symptom that will remove largest number of buckets if answered in negative.

Steps:

1. Get Bucket.
2. Find symptom with highest score in Bucket.
3. Record that symptom, score in data structure.
4. If buckets left go to step 1.
5. Find symptom with highest score in data structure
6. Return that symptom

Once the basic data is in place, DoctorSkyNet moves to the second stage and proceeds to query the patient for symptoms based on the above algorithms. It sets the 'done' variable to 1 once sufficient data about symptoms has been acquired by the system.

The overall working of DoctorSkyNet can be summarized as follows:

```
while conversation_in_progress():
    update_bucket_fractions()
    if basic_questions_unanswered():
        init_basic_questions_conversation()
```

```
if symptom_queries_not_over():
    init_symptom_query_conversation()
```

**Question Interface:** The question interface class stores the details of each question and makes it quickly accessible to any other class needing access to questions in a relatable format. It has the same attributes as any question dictionary in the data store.

**Disease Buckets:** The disease buckets are used for representing diseases. Each disease is assigned its own bucket and each bucket consists of multiple symptoms.

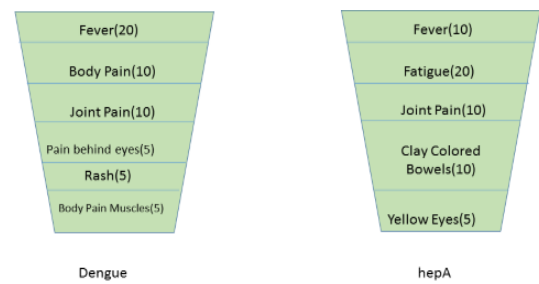


Figure 2. Sample Bucket Design

## VI. RESULTS

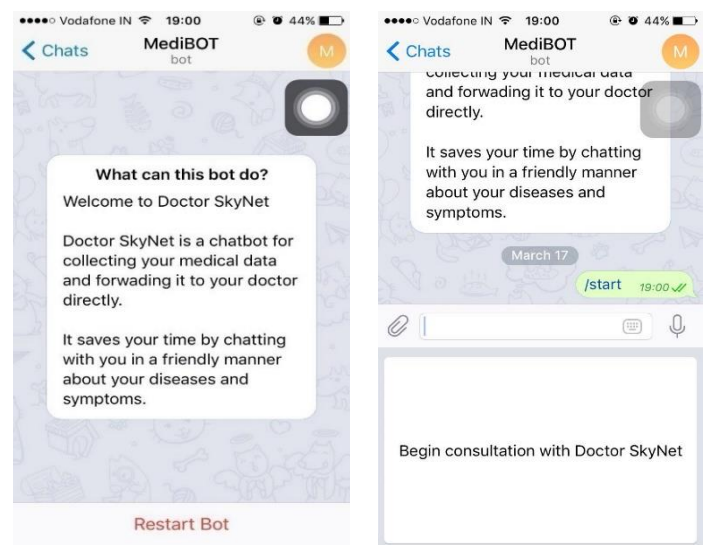


Figure 3. System UI

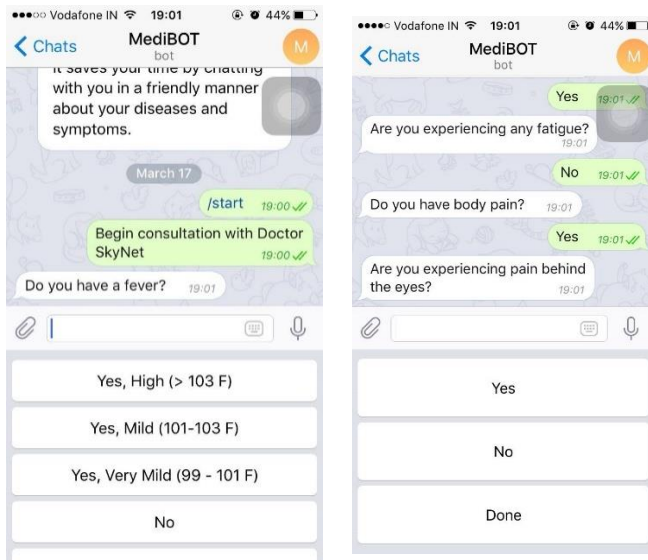


Figure 3 . System GUI

## VII. CONCLUSION

After studying various expert systems, chat bots and getting feedback from people involved in the medical profession, it can be concluded that:

- A chat bot based medical expert system can be very useful for reducing the stress that doctors have to go through on a daily basis.
- However, the results of the system should still be verified by a legitimate doctor before they are furnished to the patient.

Such a system can be perfected to incorporate all sorts of diseases

## REFERENCES

[1] Hallili, Amine. "Toward an ontology-based chatbot endowed with natural language processing and generation." *26th European Summer School in Logic, Language & Information*. 2014.

[2] Kohane, Isaac Samuel. *Temporal reasoning in medical expert systems*. Boston Univ., MA (USA), 1987.

[3] Shortliffe, Edward H., and Lawrence M. Fagan. "Expert systems research: modeling the medical decision making process." (1982).

[4] Hill, Jennifer, W. Randolph Ford, and Ingrid G. Farreras. "Real conversations with artificial intelligence: A comparison between human-human online conversations and human-chatbot conversations." *Computers in Human Behavior* 49 (2015): 245-250.

[5] Dutta, Soumitra. "Temporal reasoning in medical expert systems." *Engineering of Computer-Based Medical Systems, 1988., Proceedings of the Symposium on the*. IEEE, 1988.

[6] <http://www.wikipedia.org/>

[7] <http://www.obofoundry.org/>

[8] <https://github.com/python-telegram-bot/python-telegram-bot>

[9] <https://docs.python.org/2/>