FINAL DEGREE PROJECT

DEGREE IN INDUSTRIAL TECHNOLOGIES ENGINYEERING

PROGRAMMING AN INTELLIGENT PERSONAL ASSISTANT FOR BIOMEDICAL DATABASE ANALYSIS

REPORT

Student: Pliego Prim, Ignacio
Tutor: Perera Lluna, Alexandre
Announcement: September 2017

SUMMARY

Technology has had a great impact on society when it comes to medicine. Medical technology has been around since the first days of humanity, and since then these two terms, technology and medicine, have become much closer to each other. Their connection is increasingly useful and necessary in uncountable ways so progress in this area is unstoppable.

The purpose of this document is to establish this connection, trying to improve the relation between human and technology making it way more intuitive, comfortable and dynamic. The area chosen to work on is data analysis. This data will be biomedical as the intention is to assist medicine, making life easier for medical staff or even patients themselves. It is not the same to have to look at hundreds of data lines and analyze them manually or with the help of a software than doing the same by having a relaxed conversation with a device. What's more, anyone can have a dialogue and ask the correct questions, but few now how to work with data. So this project is a way of bringing data manipulation closer to normal people who don't have the required knowledge to treat it.

But how can this be done? As seen in the project title there is an intelligent personal assistant responsible for all the interaction human-technology. This is the Echo device from Amazon. Echo is a hands-free speaker controlled by voice. Alexa is the name for the intelligence behind Echo. Amazon provides several skills already built and also allows users to develop skills and publish them which is what’s going to happen during this project. Alexa will receive voice commands, it will understand them and match them with the operations they refer to. Then the Python code will do all the analyzing, operating and plotting required. The result will be sent on a JSON structure to the Java-code page which will interact with the Django-based web page "talktoyourdata.upc.edu". There is where the outcome will be shown as a set of blocks following the BlockChain technology. Blocks will be added, deleted, enlarged or even modified to the taste of the user, of course always inside the programmed possibilities.

The result is a interactive system where the users can take control of the actions performed on the different blocks. Finally, an interesting detail is that the outcome will be visible in any device, from laptops to phones, by only entering the web page.

INDEX

INDEX OF CODE FRAGMENTS

INDEX OF FIGURES

ABBREVIATIONS

ICU	**I**ntensive **C**are **U**nit
API	**A**pplication **P**rogram **I**nterface
UI	**U**ser **I**nterface
SOFA	**S**equential **O**rgan **F**ailure **A**ssessment
SAPS	Simplified Acute Physiology Score
JS	**J**ava**S**cript
JSON	**J**ava**S**cript **O**bject **N**otation
CNS	**C**entral **N**ervous **S**ystem
AWS	**A**mazon **W**eb **S**ervices

PREFACE

Origin and motivation of the project

Medicine is one of the most important fields in our world as humanity depends on it. Any improvement, even if it's not directly a cure to a disease, is very positive and helpful. That's the initial motivation on relating the project to medical purposes.

Writing personally, it was a big chance as I started to relate my work to future ambitions and studies, as I'm looking forward to begin the Master in Industrial Engineering specialized in Biomedicine. Also, computing and programming have always been a challenge and this project allowed me to improve my skills even more than I thought at the beginning. All together make

UPC's B2SLab has the support of different hospitals which provide real patient datasets to work with, a great agreement which benefits the two parts. This has been very useful as it gives credibility and makes the work more real.

All this mixed subjects working towards a common goal make this project possible.

Previous requirements

This project involves several areas of knowledge which are necessary to understand before starting to program.

First of all there is Alexa. It's a must to comprehend how it responds and interacts with people and all the previous work to establish the correct connection with it.
Then there is the programming side. For the most part is Python and the taught subjects in computing at ETSEIB have been really useful but there is also some Java programming that needs some learning and understanding.
Statistics are also a big part of the project. The subjects given in class and the facilities Python packages such as scikit-learn offer have been a real help.

Also, this project wouldn't have been possible without the previous work done by the B2SLab, explained later on the state of art.

1. INTRODUCTION

1.1 Objectives

The main goal trying to be accomplished in this project is to make something unknown or not understood by a collective (medical staff in this case) more reachable. Not only more reachable but also easy, comfortable and intuitive for anyone. It can be called as a way of improving ergonomics between user and computer. Medical staff such as doctors or nurses work with machines and computers which store thousands of data every day but understanding and molding it for study is not easy for them. Data analysis featuring statistics is one of the most powerful tools for comprehension, evaluation and prediction nowadays. And it's also very trustful so it's not difficult to imagine the advantages medical crew would have if they had fast and direct access to it.

Inside this major objective there is another one: demonstrating the first one by achieving a goal. As the project relates medicine and statistics this goal will include medical data analysis until finding a predictive model for future patients. Starting from a clinical dataset from the ICU, the goal will be to find the variables related with the Simplified Acute Physiology Score (SAPS) and the Sequential Organ Failure Assessment (SOFA) and then build two predictive models, each one predicting the SOFA and SAPS ratings.

1.2 The project's scope

Even though the project is faced to a specific dataset and analysis it has bigger applications. There is a IO_Block which loads the datasets and converts them into workable matrices. This block reaches more than one dataset as it can load .csv, .txt, .png, .jpg, etc. So the future intention is to be able to insert any data and work with it without caring about its type or extension. Also the purpose is to play with data any way the user wants to. It is understandable that as a final degree project working out all this is impossible so at the end this work is a demonstrator of how big this could be and opens imagination to thousands of possibilities that could be applied.

2. STATE OF THE ART

2.1 Speech recognition

Speech recognition is the ability of a device or program to identify words ans phrases in spoken language and convert them into text. The most frequent applications of speech recognition include call routing, speech-to-text processing, voice dialing and voice search. Many people think that voice recognition and talking to your devices is a technology still stuck in the past and only find it useful for asking easy questions or requesting their phone to call someone from their contacts. But nowadays that's far from truth. Although it's true this field is not as sophisticated as others may be, big companies are trying to improve it every day. At the end, prevailing depends on speed and accuracy. There is no point on using a controlled-by-voice device if the same can be done faster by typing it.

Let's see some different examples that are starting to succeed in today's market:

Ivee: smart alarm clock that turns off by telling it to shut up. It also provides some fresh morning information such as news, weather, etc.

Samsung smart TV: operable without touching any button, it allows to turn it on and off, change channels, access apps or even navigate the web by using only voice.

Skully smart helmet: as it sounds; this helmet is controlled by voice. Aiming to make motorcycle riding safer Skully is equipped with a heads-up display system with GPS navigation, Bluetooth connectivity, a full 180-degree rear view camera. All voice-controlled. No need to talk about the price, of course.

Apple's Siri: Siri is Apple's personal intelligent assistant. It's maybe the most famous voice recognition software as it is integrated in any iPhone, iPad or other Apple devices. It allows users phone and text actions, check basic information, schedule events, searching and navigating through the web, taking pictures and many more useful stuff.

Google Now: appearing in Androids with version 4.1 or higher, Google Now allows the same kind of matters as Siri. People argue that it seems to have been built with the knowledge that not a great amount of people are that keen on using a voice assistant.

Window's Cortana: It’s not just a speech assistant, but rather a Siri-like voice assistant and an intelligent text analyzer – so you can type, not just talk, to Cortana.

Ivee

Skully helmet

Figure 2.1: Examples of speech recognition devices

Amazon Echo: the device on which this project depends. Working through the on-the-cloud-intelligence Alexa, it is probably the most complete and it has the advantage of allowing users create their own skills to improve it, and improvements can be seen and used by every user as it all remains in the cloud, not in the device.

2.2 UPC's B2SLab work with Alexa

The Bioinformatics and Biomedical Signals Laboratory (B2SLab) is a research group located at the Research Center for Biomedical Engineering (CREB) of the Universitat Politècnica de Catalunya (UPC) in Barcelona, Spain. The B2SLab is focused on the analysis of bioinformatic and biomedical signals, high throughput screening (genetic association, gene expression, metabolisms) and physiological signals. This laboratory is currently lead by Dr. Alexandre Perera, tutor of this project.

At this point it is important to talk about Moritz Krügener. He is a German student who did an internship in Barcelona during four months at the B2SLab under Dr. Alexandre's tuition. Moritz started the work on the project named Complex Workflows Through Natural Language Processing, which this project aims to continue and improve. Moritz created all the complex and interactive system and this project is built on top of that. He did all the background work so that now all it takes to improve it is creating the blocks and connecting them.

Figure 2.2: B2SLab logo

3. AMAZON'S ALEXA

3.1 Alexa: the brain behind Echo

Amazon Echo, referred from now on as Echo, is a hands-free speaker controlled by voice. The device connects to the intelligent personal assistant service Alexa, which responds to the wake word "Alexa". The device is capable of voice interaction, music playback, making to-do lists, setting alarms, streaming podcasts, playing audiobooks, and providing weather, traffic and other real time information. It can also control several smart devices using itself as a home automation hub.

Examples of things that can be said to Alexa could be: "*Alexa, play Top Hits playlist on Spotify*", "*Alexa, volume up*". For it to respond is necessary to start with the wake word "Alexa".

Amazon provides users with the Alexa Skills Kit. It is a collection of self-service APIs (Application Program interface), tools, documentation and code samples that make it fast and easy for you to add skills to Alexa. All of the code runs in the cloud — nothing is installed on any user device. With the Alexa Skills Kit, you can create compelling, hands-free voice experiences by adding your own skills to Alexa. Customers can access these new skills on any Alexa-enabled device simply by asking Alexa a question or making a command.

Figure 3.1: Amazon Echo device

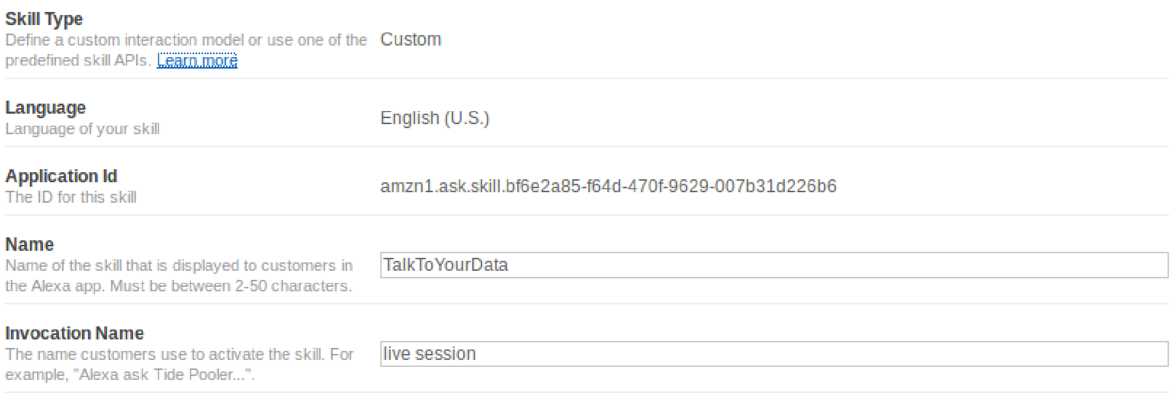
3.2 Developing the skill: "*talktoyourdata*"

New skills for Alexa are created on the website for developers from Amazon: https://developer.amazon.com. When developing a new skill Amazon provides different types. The one that's interesting for this project is the custom skill. A custom skill can handle any kind of request, as long as you can create the code to fulfill the request and provide the appropriate data in the interaction model to let users invoke the request. This is the most flexible kind of skill you can build, but also the most complex, since you need to provide the interaction model.

Let's see the steps on building the new skill:

3.2.1. Skill information:

In the skill information there are some gaps to fill in: the type, the skill name and the invocation name. This last one is how you must refer to the skill so that Alexa works in it. A good way to start is saying: "*Alexa, open live session*". This way Alexa is already inside the skill. See below that the invocation name of TalkToTourData is *live session*. Application Id is the identification given by Amazon.



Skill Type Define a custom interaction model or use one of the predefined skill APIs. Learn more	Custom
Language Language of your skill	English (U.S.)
Application Id The ID for this skill	amzn1.ask.skill.bf6e2a85-f64d-470f-9629-007b31d226b6
Name Name of the skill that is displayed to customers in the Alexa app. Must be between 2-50 characters.	TalkToYourData
Invocation Name The name customers use to activate the skill. For example, "Alexa ask Tide Pooler...".	live session

Figure 3.2: Amazon's skill information

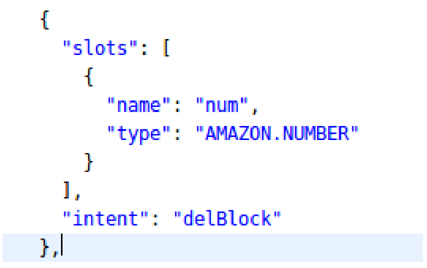
3.2.2. Interaction model:

When you create a custom skill, you implement the logic for the skill, and you also define the voice interface through which users interact with the skill. To define the voice interface, you map users' spoken inputs to the intents your cloud-based service can handle. To do this it's necessary to define the following:

- Intents: the actions that take place to fulfill the users requests. It can optionally have arguments called slots. You define the set of valid intents in a JSON structure called an intent schema.

- Slots: they are the arguments that intents optionally have. Each slot included in an intent must have a slot type. This can be either a built-in slot type or a custom slot type. Amazon provides buit-in slot types as numbers ("AMAZON.NUMBER"), dates ("AMAZON.DATES"), etc. Custom slots are made by users and must include the slot name and the values for it.

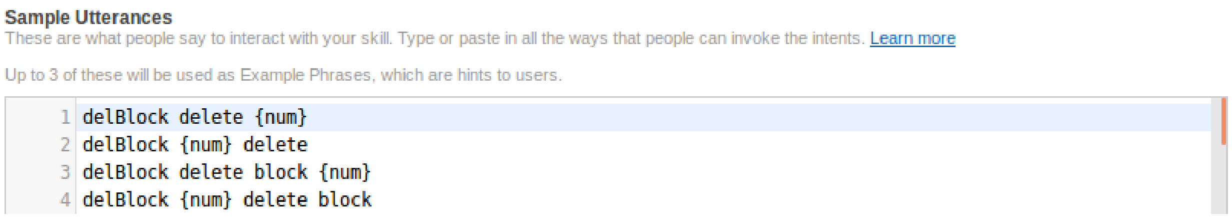
Here's an example of an intent in JSON structure. The "intent" property gives the name of the intent; "delBlock" in this case. The "slots" property lists the slots associated to the intent. In this case there is only one: "num" which refers at the built-in slot "AMAZON.NUMBER".



```
{
  "slots": [
    {
      "name": "num",
      "type": "AMAZON.NUMBER"
    }
  ],
  "intent": "delBlock"
},
```

Figure 3.3 Intent and slot example

- Sample utterances: they are all the spoken phrases assigned to the intents. An intent can have as many utterances as the user wants, and seems obvious that the more it has the easier communication with Alexa becomes.
Examples of utterances for the intent *delBlock* could be: "delete block {*num*}", "delete {*num*}", "{*num*} delete", "block {*num*} eliminate", etc. All this phrases call the *delBlock* intent when spoken.
To map utterances with intent the following method has to be used:
· Name of the intent on the left
· The phrase a user might speak on the right


Sample Utterances
These are what people say to interact with your skill. Type or paste in all the ways that people can invoke the intents. Learn more

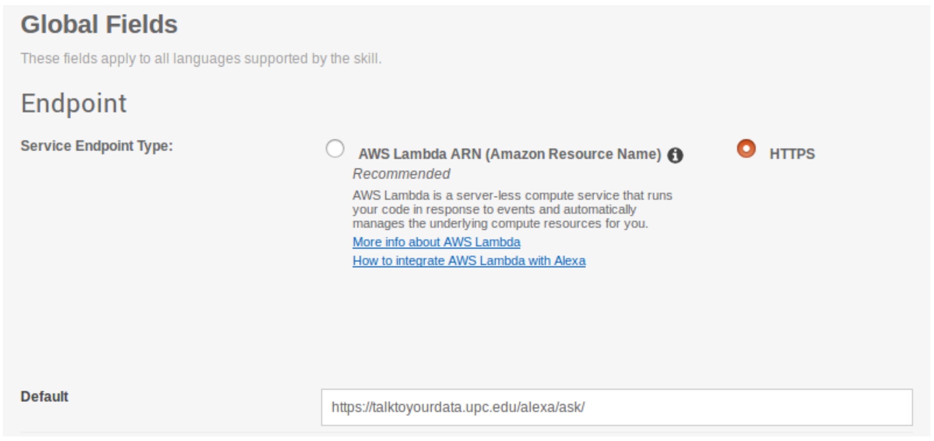
Up to 3 of these will be used as Example Phrases, which are hints to users.

```
delBlock delete {num}
delBlock {num} delete
delBlock delete block {num}
delBlock {num} delete block
```

Figure 3.4: Sample utterances for delBlock

3.2.3. Configuration:

Amazon let users decide whether they prefer using AWS Lambda or HTTPS. The easiest way to build a custom skill is by using AWS Lambda as it executes the code only when needed and there is no need to run servers every time. But in this project AWS Lambda is not enough so the skill is hosted as a web service. The chosen domain is one bought by UPC's B2SLab: https://talktoyourdata.upc.edu/alexa/ask.


Global Fields

These fields apply to all languages supported by the skill.

Endpoint

Service Endpoint Type: AWS Lambda ARN (Amazon Resource Name) [✓ HTTPS]

Recommended

AWS Lambda is a server-less compute service that runs your code in response to events and automatically manages the underlying compute resources for you.

More info about AWS Lambda

How to integrate AWS Lambda with Alexa

Default: https://talktoyourdata.upc.edu/alexa/ask/

Figure 3.5: Amazon's endpoint configuration

4. METHODOLOGY

4.1 Overview

The process from speaking a phrase till having the web giving a response follows a few steps:

1. Relating the different *intents* with the functions they must activate, in `Alexa.py` or handling UI inputs from the browser with the `consumers.py`.

2. Once inside the intent, navigating through the BlockChain with the help of `BlockChain.py`.

3. Creating the block in a Python class format. Each block is a different class, but all of them inherit from the *BaseBlock* class, in `BaseBlock.py`. This step doesn't always happen as some intents don't give way to creating a block, for example the *delBlock*.

4. Sending the block information via JSON structure to a Java file (`live.html`). The JSON is created in each class in the `getNode()` function.

5. Trough `live.html` create the interaction with the web.

6. Watching the result in the web page (https://talktoyourdata.upc.edu).

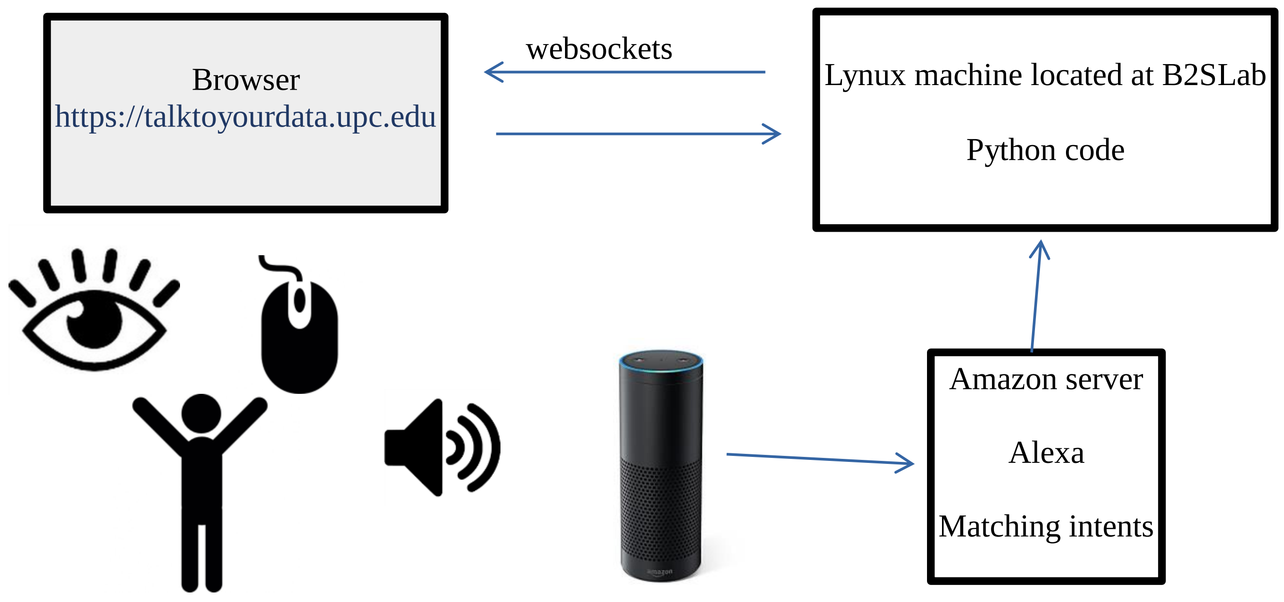


Figure 4.1: Process representation

4.2 Alexa.py

This Python file is the only one that interacts directly with Alexa. This file is where all Alexa intents and utterances are defined. The django app has a rest API endpoint at /alexa/ask/ that will handle all the incoming request and route them to the intents defined in any `Alexa.py` file. Each intent must return a valid Alexa response dictionary. To aid in this the django-alexa API provides a helper class called *ResponseBuilder*. This class has a function to speed up building these dictionaries for responses.
To clarify the explanation will be based on an example; the intent that enlarges the image in a block. Before starting some things must be imported.

```
from django_alexa.api import fields, intent, ResponseBuilder
```

```

class Num(fields.Amazon.slots):
        num = fields.Amazon.Number()
        #this class creates a custom slot named "num" and type Amazon.Number


@intent(slots=Num, app="AlexaHandler")
def showImg(session, num=0):
        """
                                Showing block with number X in new tab
                                ---
                                Show block {num}
                                show {num}
        {num} show
        show              #list of sample utterances that the user can say
        show larger {num}
        {num} show larger
        show larger
        larger
        enlarge
                """
        SessChain = consumers.getSessChain() # gets the actual BlockChain session
        if type(num) is int:
                                if num < SessChain.getBlockListLength():
                msg = "showing block " + str(num)
                try:
                SessChain.getBlock(num).showBlock(num)


                                                #gets the block nunber and calls the showBlock function if it is
                                available in the class.
                except:
                msg = "function not available for block " + str(num)
        else:
                msg = "Block with number " + str(num) + " does not exist. Maximum block number is " +
str(SessChain.getBlockListLength() - 1)
                else:
                                print("\033[94m Amazon provided " + str(type(num)) + " type \033[0m")
        try:
                SessChain.getBlock(-1).showBlock()
                msg = "showing last block"


                        #gets the last block and calls the showBlock function if its available
        except:
                msg = "Function not available for last block"


        return ResponseBuilder.create_response(message=msg,
                     reprompt="",
                     end_session=False,
                     launched=True)


                                #ResponseBuilder generates the reponse Alexa gives the user; the message. Reprompt is

                what Alexa says in case the user remains silent.
```

Code fragment 4.1: Intent example: showImg

All the intents have this structure. Once the class Num is created, the slot num is available for any intent. But what if the slots in the utterances are not predefined in fields.Amazon? Then the user must create them. This happens for example in the intent *passOption*, and it works like this:

```
OPTIONS = ["sand", "glass", "void", "histogram", "regression", “regression line”]

class OPT(fields.AmazonSlots):
        alexa_option = fields.AmazonCustom(label="OPTIONS", choices=OPTIONS)
        number = fields.AmazonNumber()

@intent(slots=OPT, app="AlexaHandler")
def passOption(session, number=-1, alexa_option=""):
        """
        option passing
                        ---
                        pass option {alexa_option} to block {number}
                        option {alexa_option} block {number}
                        block {number} option {alexa_option}
        """
        .
        .
        .
        .
```

Code fragment 4.2 Intent with slot defining

The *alexa_option* slot gets its possible words from the new list of strings "OPTIONS". So an example of utterance could be: "*Alexa, pass option histogram to block 5"*.

The django-alexa framework also provides two django management commands that will build intents and utterances schema for the user by inspecting the code.
In the showImg intent:

```
>>> python manage.py alexa_intents
 {
    "slots": [
      {
        "name": "alexa_option",
        "type": "OPTIONS"
      },
      {
        "name": "number",
        "type": "AMAZON.NUMBER"
      }
    ],
    "intent": "passOption"
 },
```

Code fragment 4.3: Alexa intents

```
>>> python manage.py alexa_utterances

showImg show block {num}
showImg show {num}
showImg {num} show
showImg show
showImg show larger {num}
showImg {num} show larger
showImg show larger
showImg larger
showImg enlarge
```

Code fragment 4.4: Alexa utterances

4.3 Consumers.py

The consumers.py mission is to handle the incoming socket messages with its three functions ws_add, ws_message and ws_disconnect. As the names imply, they get called on client connection events and incoming messages. ws_message has to handle most of the messages, mainly it handles UI input directly from a client browser coming in as a JSON and manipulating the BlockChain accordingly. In general, any function anywhere on the server can broadcast to the Group channel and distribute messages, so consumers.py is only really important for new clients and incoming new IU traffic.

If a message should be sent from anywhere, import the Group. That means everyone inside the https://talkyoyourdata.upc.edu/ can contemplate action at the same time; live.

```
from channels import Group
# data has to be a JSON file adhering to the protocol
# raw JSON
Group("alexa").send({
                "text": json.dumps(data)
})
# block data
Group("alexa").send({
                "text": block.GetNode()
})
```

Code fragment 4.5 Group function

consumers.py also includes an important function that gets the actual session and identifies the BlockChain to work on it. It's the getSessChain():

```
def getSessChain():
    global SessChain
    if SessChain != "init":
        return SessChain
    else:
        try:
            print("getSessChain loading:")
            oldT = time.time()
            SessChain = cache.get("alexa")
            print("done after: ", (time.time() - oldT), "seconds")
            return SessChain
        except NameError("getSessChain Error"):
            print("\033[91m SessChain currently not defined \033[0m")
```

Code fragment 4.6: getSessChain function

So to work on the current *BlockChain* a variable SessChain = consumers.getSessChain() must be added. This *SessChain* contains the actual chain. Once in the current chain actions from the *BlockChain* class in BlockChain.py can be performed on *SessChain*, such as Sesschain.getBlock(), SessChain.getBlockListLength(), etc.

As said, ws_message handles UI input directly from the browser, i.e. for example pressing buttons. It is discouraged to use the buttons but some of them have been implemented and work as a command. Below an example of clicking the button "boxplot", which creates a boxplot block.

```
SessChain = consumers.getSessChain()
elif data['cmd'] == "click":                                    # for clicking commands enters here
                elif "boxplot" in str(data['opt']):             # looks if "boxplot" is the button's name
                    block = SessChain.getBlock(data['num'])
                    data = block.getData()
                    Box = Boxplot(data=data, session="alexa", name="Boxplot")
                    SessChain.addBlock(Box)
                    Group("alexa").send({
                        "text": Box.GetNode()                   # sending the info to all clients
                    })
```

Code fragment 4.7: "Click" command functioning

4.4 BlockChain

The way blocks are displayed and connect follow the BlockChain technology. Together with ts concept of "distributed ledger technologies, Blockchain is presented as a set of technologies (P2P, time stamping, cryptography, etc.) that combine to make it possible for computers and other devices to manage their information by sharing a distributed, decentralized and synchronized registry of all of them, instead os using the traditional databases. But not only that; the information is transmitted and stored in an extremely secure way respecting identity and privacy thanks to the use of cryptographic keys. It also doesn't allow alteration, i.e. undoing or rewriting off the already registered, which is also visible to any participant in the network if it's public, adding a great transparency. The most known example is Bitcoin.

4.4.1 BlockChain.py

On the BlockChain.py file basic functions for it to work out are implemented in the BlockChain class.

```
class BlockChain:
    def __init__ (self, name="", session=""):
        ''' initial save '''
    ...
    def addBlock (self, Block):
        ''' add Block to BlockList, add Vars from Block to VarList, add Connections '''
    def getBlock (self, index)
    def getBlockList (self)
    def getBlockListLength (self)
            ‘” returns the number of blocks ‘”
    def delBlockByIndex (self, index)
                        ‘” deletes block by sayin its number ‘”
    def delBlockByElement (self, Block):
        ''' del associated elements in VarList->call Block.delBlock() for cleanup of cache data ‘”
    def getBlockId (self, Block)
    def delBlocksAll(self):
        ''' reset everything but keep Session alive '''
    ...
    def Chain_pickle (self):
        ''' pickle self in cache folder '''
    def __str__(self):
        ''' provide summary of status '''
```

Code fragment 4.8: BlockChain.py

4.5 Blocks

4.5.1 Common functions

Each block (class) has it's own functions and the only contact with other blocks comes with the getData function. But there are some functions that appear in more than one class and are interesting to study individually.

4.5.1.1 GetNode(self):

The GetNode function it's really important ans shows up in every block with no exception. It is in charge of storing the block information and content and send it in a JSON structure to the Javascript file live.html. Two examples for it's structure are:

```
def GetNode(self):
    print("get IO Node")
    IO_data = ["Variable Name:", self.vars, "Dimensions:", str(len(self.data))+ "x"+ str(len(self.data[0])), "Type:", self.display_type]
    data = {"type": "block",
            "block_type": self.type,
            "block_num": self.block_num,
            "file_name": self.file_name,
            "content_type": "IO_data",
            "IO_data": IO_data,
            "options": self.options,
            "vars": self.vars,
            "call_path": self.call_path,
            }
    return json.dumps(data)
```

Code fragment 4.9: GetNode() function for text

Above the structure for a text information storing block. It creates a python dictionary and sends it as a JSON. The information provided is vital to separate and treat the block once in the live.html.

```
def GetNode(self):
    call_path = settings.CACHE_URL + "/" + self.session + "/" + self.name + '.png'
    data = {"type": "block",
            "block_type": self.type,
            "block_num": self.block_num,
            "file_name": self.file_name,
            "content_type": "image",
            "call_path": call_path,
            "options": self.options,
            "vars": self.vars,
            "update": self.update,
             }
    return json.dumps(data)
```

Code fragment 4.10: GetNode() function for images

Above the structure for an image showing block. The call_path gives the image location in the server (CACHE_URL = "https://talkyotourdata.upc.edu/AlexaHandler/cache") or in the local environment (CACHE_URL = "http://localhost:8000/AlexaHandler/cache") for working in localhost. Changing the work environment is done by settling ENV variable to "SERVER" or "LOCAL" in the settings.py.

4.5.1.2 getData(self):

The getData function is the responsible for communication between blocks. A block that follows another gets information from it trough getData. Below two examples of the structure. They consist in Python dictionaries with their keys attached to the blocks information.

```
def getData(self):

   data = {"name": self.data_name, "type": self.file_type, "data": self.data, "titles":self.titles}
   return data


def getData(self):


   data = {"name":self.name, "type": self.type, "data": self.train, "train_data": self.train, "test_data": self.test, "titles":self.titles,
"vars":self.vars}


   return data
```

Code fragment 4.11: getData() function

The point of this function is to store any data, variable or information of any type which is going to be used by other block. All blocks have it except those ones that are at the end of the chain; no one gets information from them.

4.5.1.3 showBlock(self, num=""):

This function only works for images. Its duty is to enlarge them so the user has a better visual experience. To activate it the word "show" should be enough, and to return it to the initial size all it takes is the word "minimize".

```
def showBlock(self, num=""):


   call_path = settings.CACHE_URL + "/" + self.session + "/" + self.name + '.png'
   data = {"type": "cmd",
         "block_num": num,
         "cmd": "show",
         "call_path": call_path,
         }
   print("executing showImageBlock")
   Group("alexa").send({
      "text": json.dumps(data)
   })
```

Code fragment 4.12: showBlock() function

The showBlock function requires a number that indicates which block has to be enlarged. If no number is given it will perform on the las one if it has the property.

4.5.1.4 getOption(self, number, para):

If a Block accepts to be called with options via the *passOption* intent, it has to provide a `getOption` function that deals with the incoming option string. This function is called by saying, for example, *"Alexa, option regression, block 3"*. In this case, blocks may themselves create more blocks and add them to the BlockChain or update the existing block. After adding a Block, it is necessary to distribute the new display information to all clients, through the `Group.send`. The inputs for the function are the option (para) and the block number (number).

```
def getOption(self, para, number):
        SessChain = consumers.getSessChain()
        .
        .                   #coding for a new block
        .
        .
        self.name = self.name + '.'+para
        self.cache_path = settings.CACHE_DIR + "/" + self.session + "/" +         self.name + '.png'
        self.options = ["show"]
        self.update = "true"                #updating the new atributes
        self.block_num = number
        Group("alexa").send({                       #sending the update to all clients, in this case the
          "text": self.GetNode()            new JSON provided by GetNode()
        })
        # reset state
        self.update = "false"               #setting update to initial value again
        SessChain.Chain_pickle()            #saving the session
```
Code fragment 4.13: getOption() function

Every *getOption* function must include a `self.update` attribute. When the block is first created its value must be set to "false". If the state of the block is to be changed it must be set to "true", then send the information to all clients and finally set it to "false" again. It is important as live.html starts by checking: `if (data.update != "true") {`, and it treats different new blocks from updated ones, for example setting the correct block numbers for them.
This way, a message block can turn into an image block or vice versa, or just an updated image. The content of the block is loaded as if it was a new block. Options and variables can also be changed by changing `self.options` and `self.vars`. An implementetion of this can be found for example in the `LinearRegressionBlock` in the `getOption`.

4.5.2 BaseBlock.py

As said before, each block is a Python class that has it's own functions. There are blocks designed to only store data, others to provide information through messages and others to show images as graphs. But they all have one thing in common: they inherit from the class BaseBlock:

```
class BaseBlock:
	def__init__(self,name="base",type="base",session="", content_type=""):
		self.name = name
		self.type = type
		self.session = session
		self.content_type = content_type
		self.options = []
		self.vars = []
		self.block_num = ""
	def getName(self):
		return self.name
	def setName(self, name):
		self.name = name
	def getSession(self):
		return self.session
	def setSession(self, session):
		self.name = session
	def getType(self):
		return self.type
	def setType(self, type):
		self.type = type
	def getOptions(self):
		return self.options
	def addOption(self, s):
		self.options.append(s)
	def GetNode(self):
		import json
		data = {"type": "block",
		    "block_type": self.type,
		    "block_num": self.block_num,
		    "content_type": self.content_type,
		    "options": self.options,
		    "vars": self.vars,
		    }
		return json.dumps(data)
	def delBlock(self):
		del self
	def getData(self):
```

Code fragment 4.14 BaseBlock.py

The Baseblock gives any block the basic attributes which can be changed later in the blocks themselves if necessary.

4.5.3 Functional blocks

4.5.3.1 Main structure

The main structure of any block is the following:

```
class ClassName(BaseBlock):

        def __init__(self):
                self.name = name
        self.session = session                          #defines all the attributes
        self.block_num = ""
        self.type = "matrix"
        self.options = ["option1","option2","option3"]
        self.vars = []
                self.update = "false"
                .
                .
                .               #where the block is created
                .

        def getData(self):
                .....
                return data

        def GetNode(self):
                data = {"type": "block",
                "block_type": self.type,
                "block_num": self.block_num,
                "content_type": "content",
                "data": data,
                        "call_path": settings.CACHE_URL + "/" + self.session +"/" + self.name,
                "options": self.options,
                "vars": self.vars,
                    }
                return json.dumps(data)


        def showBlock(self, num=""):    #if the block contains an image
                call_path = settings.CACHE_URL + "/" + self.session + "/" +self.name
                data = {"type": "cmd",
             "block_num": num,
                  "cmd": "command",
             "call_path": call_path,
                  }
                Group("alexa").send({
                  "text": json.dumps(data)
                })

        def getOption(self, parameters):  # if necessary
                SessChain = consumers.getSessChain()
                .
                .                       # updating the block
                .
                .
                self.name = self.name + '.'+para
                self.cache_path = settings.CACHE_DIR + "/" + self.session + "/" + self.name
                self.options = ["show"]
                self.update = "true"                            # gives new values to attributes
                self.block_num = number
                Group("alexa").send({
                  "text": self.GetNode()  # sending modifications to everyone
                        })
                # reset state
                self.update = "false"
                SessChain.Chain_pickle()        #saving the session


        def AnyOtherFunction():         # other functions that the block may require
                .
                .
                .
```

Code fragment 4.15: Main structure of a class

4.5.3.2 IO_Block.py

The IO_Block is the first one in use as it's the one in charge of loading files. At the moment it can load text files (.csv, .txt, .dat) or images (.png, .jpg, .jpeg). When loading data in text format, it uses the numpy function np.genfromtext(data) which displays the data as a matrix where each line is a observation and each column is a variable. This matrix is ready for manipulation. For images it uses pl.imread(path) from matplotlib.pylab. All the files to be loaded are stored in the /import folder.

```
class IO_Block(BaseBlock):
    def __init__(self, file_name, name="IO", session="", abs_path=""):
        self.name = name
        self.type = "IO"
        self.session = session
        self.vars = []
        self.file_name = file_name
        self.data_name = ".".join(self.file_name.split(".")[:-1])
        self.file_type = ".".join(self.file_name.split(".")[-1:])
        self.display_type = ""
        self.call_path = ""
        self.block_num = ""
        self.options = []
        if abs_path == "":
            self.path = settings.IMPORT_DIR + "/" + self.file_name
        else:
            self.path = abs_path
        if self.file_type in ["csv", "txt", "dat"]:
            self.data = np.genfromtxt(self.path, delimiter=",", dtype=None, names=True)
            self.titles = self.data.dtype.names
            self.display_type = "matrix"
            self.type = "matrix"
        elif self.file_type in ["png", "jpg", "jpeg"]:
            self.data = pl.imread(self.path)
            self.display_type = "image"
            self.type = "image"
            self.cache_path = settings.CACHE_DIR + "/" + self.session + "/" + self.file_name
            self.call_path = settings.CACHE_URL + "/" + self.session + "/" + self.file_name
            try:
                copyfile(self.path, self.cache_path)
            except:
                print("\033[93m Errror importing:", sys.exc_info(), "\033[0m")
        else:
            raise NameError("unsupported format")


                def showBlock(self, num=""):
                                # displays if it's an image
                def getData(self):
                                data={"name":self.data_name,"type":self.file_type,"data":self.data, "titles":self.titles}
                                return data
                def getTitles(self):
                                return self.titles  #all the variable names
                def GetNode(self):
                                IO_data = ["Variable Name:", self.vars, "Dimensions:", str(len(self.data))+ "x"+ str(len(self.data[0])), "Type:",
self.display_type]
                                data = {"type": "block",
                                "block_type": self.type,
                                "block_num": self.block_num,
                                "file_name": self.file_name,
                                "content_type": "IO_data",
                                "IO_data": IO_data,
                                "options": self.options,
                                "vars": self.vars,
                                "call_path": self.call_path,
                                }
                                return json.dumps(data)
```

Code fragment 4.16: IO_Block.py. Example of a complete class.

4.5.3.3 TrainTestBlock.py

This block's mission is to separate the data into two sets. One for training data and the other for testing. The goal is to analyze the train data, build a predictive model and test the results with the test data.

The split is done with the `sklearn.model_selection` package function `train_test_split`, which divides arrays and matrices into random train and test subsets:

```
self.train, self.test = train_test_split(self.data, train_size=u)
```

The data is taken from the `getData()` function in the *IO_Block*, and `train_size` is a parameter that decides the percentage of data stored in the train subset; and consequently the stored in the test subset.

The `__init__()` includes some code lines that look in every variable and separate the binary ones from the not binary. After, it stores them in the `self.vars` list which will be displayed on the UI; and knowing the category of the variable is helpful when asking for actions to perform on them.

The other functions available in this block are `getData()` and `getNode()`.

4.5.3.4 StatisticsBlock.py

The statistics block receives two inputs: data and parameter. The data is taken from a IO_Block or a TrainTestBlock. The parameter must be a number related to a variable. This variables in the code are called "titles", and its a list of all the variable title names. For calculating statistics the function library used is `statistics.py`. The information provided by the block is:

· Mean: `stat.mean()`
· Median: `stat.median()`
· Mode: `stat.mode()`
· Standard deviation: `stat.pstdev()`
· Variance: `stat.pvariance()`
· Maximum value: `max()`
· Minimum value: `min()`

The other functions available are `getData()` and `getNode()`.

4.5.3.5 Histogramblock.py

As seen in its name, this block creates and displays histogram images. The inputs necessary are the data and a parameter. The histograms are created with the functions available in the `matplotlib.pyplot` library. Imported as plot, the ones used are: `plot.hist()` for doing the histogram, `plot.xlabel()` to name the abscissa, `plot.savefig()` to save it in the `/cache/alexa` folder and `plot.close()` for closing it.

The parameter can have two different type of values and because of that there are two ways of creating a Histogram:

1. If parameter is a `str` (it gets "default" as the default value): the previous block must be a *StatisticsBlock*. A number is not required as it takes the one from the block and makes the plot. At the moment it plots the mean,but it is easy to implement another type of histogram.

2. If parameter is a `int` (number): the previous block must be a *IO_Block* or a *TrainTestBlock*. It gets the number and makes the plot of this variable.

Its other functions are `getNode()`, `getData()` and `showBlock()`.

4.5.3.6 Boxplotblock.py

Obviously for creating Boxplot diagrams. It works similar to the HistogramBlock, having the data and parameter as inputs. It uses the matplotlib.pyplot library and the functions called are: `plt.figure()`, `plt.add_subplot()`, `plt.boxplot()`, `plt.set_xlabel`, `plt.savefig()` and `plt.close()`.

There are two ways of creating the BoxplotBlock and it works equal to the HistogramBlock.

4.5.3.7 LinearRegressionBlock.py

This block consists of two parts. First one draws a scatter plot with two variables. The second one uses the `getOption()` function to update the block and shows the same scatter plot but with the regression line and gives the Pearson correlation.

It uses the following libraries: `matplotlib.pyplot` for drawing the graph, `numpy` for finding the regression line and `scipy.stat.stat` for calculating the Pearson correlation value.

4.5.3.8 choosecolumnBlock.py

This block's only function is to store the data from the columns the user wants, making them easy to be used afterwards. It includes a SOFA exception that selects automatically the SOFA-related columns.

4.5.3.9 RegressionBlock.py

This block build a linear regression model with the given variables. At the moment it only takes data from the choosecolumnBlock.

4.5.3.10 outliersBlock.py

Deletes the outliers from the data. A way of cleaning it and making the results more coherent.

4.5.3.11 SOFABlock.py

Estimates the SOFA score value for a specific patient.

4.6 live.html

The live.html is the responsible file for interacting directly with the web. It is a JavaScript file and is in charge of handling all the web structure, creating the web style and all the visuals. It also handles all the incoming messages from the blocks, i.e. the JSONs with the block information.

Commands are delivered sequentially in individual socket messages. All incoming socket traffic is handled by the `socket.onmessage` function and routed to functions such as `new_grid_block(data)` for further processing.

```
socket.onmessage = function(e){
    data = JSON.parse(e.data);
    // routing
    if(data.type == "cmd")
        // route commands
        if(data.cmd == "ready")
            socket.send(JSON.stringify({"type": "cmd", "cmd": "init"}))
        else if (data.cmd == "file_list")
            // handling the UI fileList
        else if (data.cmd == "reset")
            // reset UI
        else if (data.cmd == "del_block")
            // delete block from UI
        else if (data.cmd == "show")
            // animate block image zoom
        else if (data.cmd == "listening")
            // indicate active Alexa session
        // etc.
    else if(data.type == "block")
        new_grid_block(data);
}
```
Code fragment 4.17: socket.onmessage function

New UI elements are handled by `JavaScript`. If for example a new block carries the `var` keyword in it's `JSON`, everything in the list is added appropriately to the left-hand `varList`. Same goes for all `opt` entries, that are generated as grey buttons for any incoming block.

Block `div` id's follow the naming scheme `grid_block+block_number` with class `content`, `mdl-cell` and it's `mdl` options. The `div` can be filled with the content such as `<img>` tags or `<p>`. Paragraphs should get the `Info` css-class for unified display style. Width of the block can be influenced by the `mdl-cell--X-col` option replacing `X` with a number between 1-12. Below this, the `button-group div` is rendered automatically.

```
<div id="grid_block0"> class="content mdl-cell mdl-cell--middle mdl-shadow--6dp mdl-cell--5-col">
    <img src="cache_url">
    <p class="Info mdl-shadow--6dp">
    Info text
    </p>
    <div id="button_group" class="button_group">...
    </div>
</div>
```
Code fragment 4.18: <div> creation example

Block creation is done via JavaScript only:

```
function new_grid_block(data){

    // ABBREVIATED AND SIMPLIFIED function, see source
    // get parent and create new block div
    var parent = document.getElementById("grid");
    //update happens also in this funciton
    if(data.update != "true"){
        name = "grid_block" + grid_count;
        var div = document.createElement("div");
        div.id = name;
    }
    else{
        name = "grid_block" + grid_count;
        var div = document.getElementById(name);
        div.innerHTML = "";
    }
    div.className = "content mdl-cell mdl-cell--middle mdl-shadow--6dp";

    // add variables to list and jquery Data
    if(data.hasOwnProperty("vars")){
            new_Var(data.vars, grid_count);
            jQuery.data( div, "vars", {
               vars: data.vars
            });

    // specific block_type, it creates the block depending on if its an image, text, etc..
    if(data.block_type == "rich_image"){
        var img = document.createElement("img");
        img.src = data.call_path;
        var p = document.createElement("p");
        p.className = "Info mdl-shadow--6dp";
        p.appendChild(document.createTextNode("Info: " + data.add_data));
        div.appendChild(img);
        div.appendChild(p);
        div.className += " " + "mdl-cell--5-col"
    }
    // else if other types
    // ...

    // automatic button-group creation
    // ...
    div.appendChild(but_group);
    if (data.update != "true"){
        grid_count +=1;
        $(parent).append(div);
    }
    // add full block to grid
    // screen size and loading handling here
    // ...
}
```

Code fragment 4.19: new_grid_block() function

5. RESULTS

5.1 Voice user interface

One of the chased results of this project is to be able to speak to Alexa and obtain visual and verbal outcomes. All the phrases a user can say are the utterances defined and the outputs will rely on the intents to which they are matched. The verbal outcomes are defined in the `Alexa.py` file and stored in the `msg` variable (see code fragment 4.1).

5.1.1 Table of intents

The following table shows the intents existing now, its use and the matched utterances:

INTENT	**ABILITY**	**UTTERANCES**
SessionEndedRequest	Ends the session	*end* *quit*
showImg	Enlarges the image in the block	*show block {num}* *show {num}* *{num} show* *show* *show larger block {num}* *show larger {num}* *{num} show larger* *show larger* *larger* *enlarge* *enlarge block {num}* *enlarge {num}* *{num} enlarge*

loadFile	loads different types of files turning them into workable matrices	*read file {num}*
		read {num}
		{num} read
		load file {num}
		load {num}
		{num} load
		file {num}
		{num} file

passOption	Calls the getOption function of the block, allowing modifications in it	*pass option {alexa_option} to block {number}*
		option {alexa_option} block {number}
		block {number} option {alexa_option}

INTENT	**ABILITY**	**UTTERANCES**
TrainTest	Separates train from test data	*block {num} separate train data with train size {train_size} per cent* *block {num} choose train data with train size {train_size} per cent* *block {num} separate train data from test with {train_size}* *separate train data from test* *separate test from train* *separate train from test* *separate test data from train*

Boxplot	Makes box plot graph	*Block {num} boxplot with variable {var}*
		Boxplot block {num} variable {var}
		Boxplot block {num} with variable {var}
		Boxplot block {num} column {var}
		Boxplot
		Create boxplot
		Do boxplot with variable {var}
		Do boxplot with column {var}
		Block {num} boxplot
		Boxplot block {num}

Statistics	Creates an information block showing some descriptive statistics	*Block {num} show statistics for column {col}* *Block {num} show statistics for {col}* *Show statistics for {col}* *Show statistics* *Block {num} do statistics* *Do statistics to block {num}* *Do statistics*

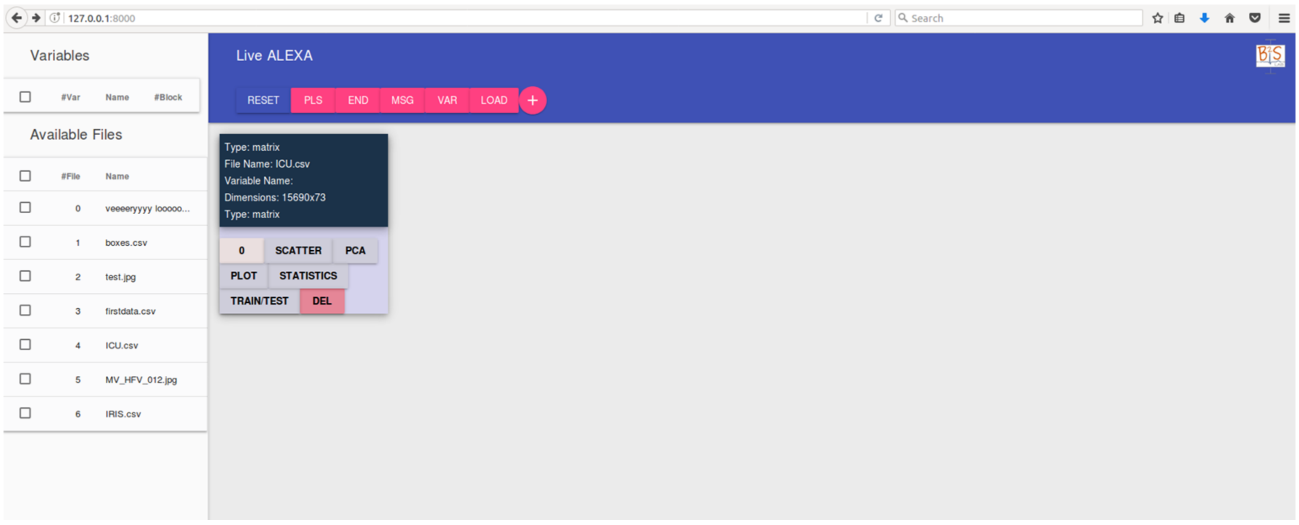
INTENT	**ABILITY**	**UTTERANCES**
minimize	Minimizes after showImg	*minimize* *make small* *hide* *back*
Scatterplot	Draws a scatterplot for two variables	*Block {num} draw a scatterplot for variables {num1} and {num2}* *Block {num} draw a scatterplot for {num1} and {num2}* *{num1} and {num2} scatterplot* *Make a scatterplot for {num1} and {num2}* *{num1} {num2} scatterplot*

delBlock	Deletes the block	*delete {num}*
		{num} delete
		delete block {num}
		{num} delete block

choosecolumnBlock	Chooses the columns to store in the block	*choose columns {num1}.....* *choose columns for sofa score* *choose columns for sofa*
RegressionBlock	Build a linear regression model	*build a regression model* *block {num} regression model*
outliersBlock	deletes the outliers rows	*clear datab* *clean data* *block {num} clean* *delete outliers* *block {num} delete ouliers*
SOFABlock	calculates the sofa score with the formula taken from the RegressionBlock	*what is the sofa score for patient {num}* *sofa score for patient {num}* *patient {num} sofa score* *predict the sofa score for {num}* *predict the sofa score for patient {num}*

5.2 Web user interface

5.2.1 Main view



#Var	Name	#Block

#File	Name
0	veeeeryyyy looooo...
1	boxes.csv
2	test.jpg
3	firstdata.csv
4	ICU.csv
5	MV_HFV_012.jpg
6	IRIS.csv

```
Type: matrix
File Name: ICU.csv
Variable Name:
Dimensions: 15690x73
Type: matrix
```

Figure 5.1 Main UI view

Shown above is the user interface; where all the results from interaction with the browser or Alexa are visualized. In the image the server is running in localhost:8000 for testing purposes, but the web page where the results are showed is https://talktoyourdata.upc.edu/alexa/ask, as Amazon is currently pointed towards https://yourURL.com/alexa/ask to send all the Alexa requests. For changing where the server runs the only thing to do is modify the ENV variable to "LOCAL" or "SERVER" in settings.py.

For getting the web working Django must be running, and that is done through the command `>>>python manage.py runserver`

Before entering a user name and password are required for entry as there are functioning buttons inside which everyone can use.

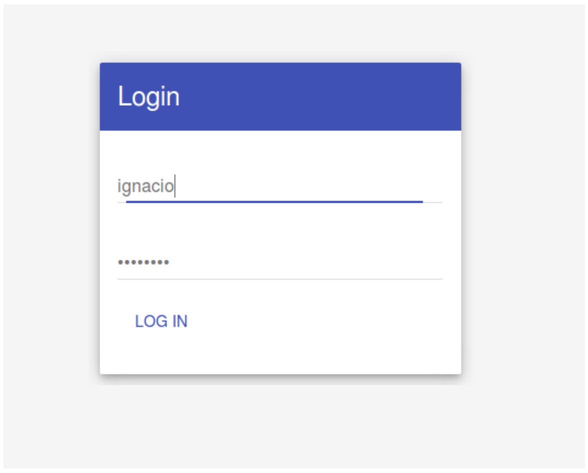
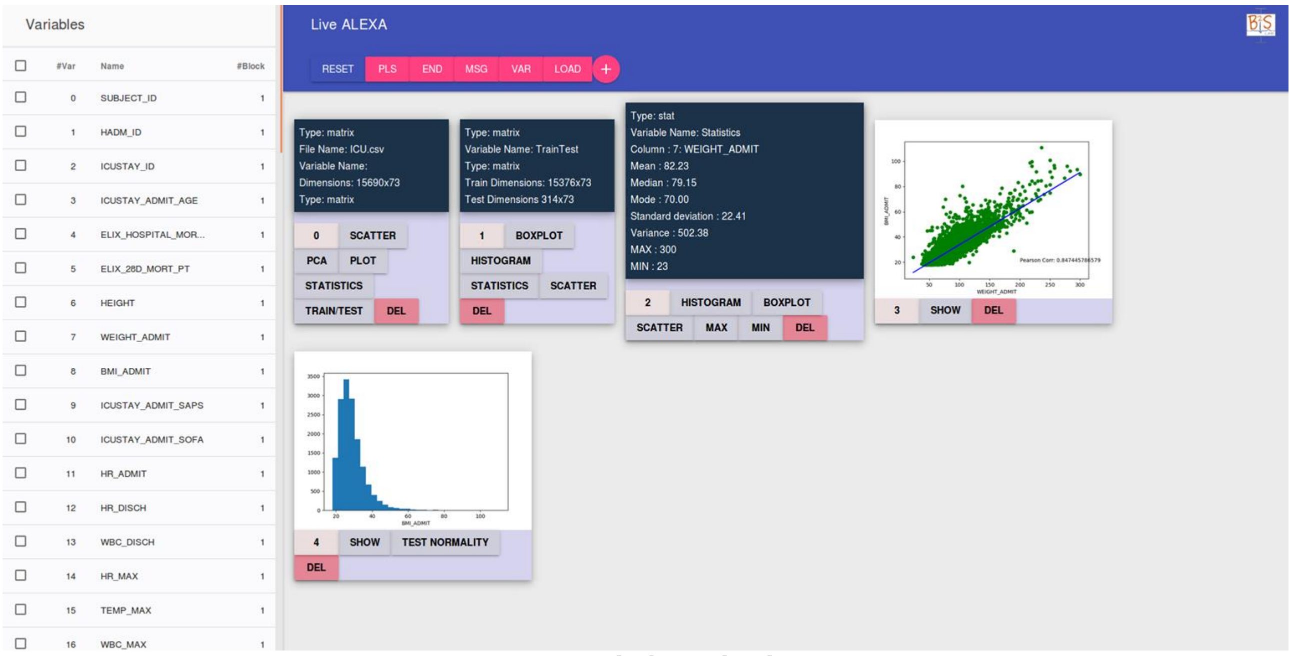


Figure 5.2 Login rectangle

The UI consists of three main elements:



	#Var	Name	#Block
☐	0	SUBJECT_ID	1
☐	1	HADM_ID	1
☐	2	ICUSTAY_ID	1
☐	3	ICUSTAY_ADMIT_AGE	1
☐	4	ELIX_HOSPITAL_MOR...	1
☐	5	ELIX_28D_MORT_PT	1
☐	6	HEIGHT	1
☐	7	WEIGHT_ADMIT	1
☐	8	BMI_ADMIT	1
☐	9	ICUSTAY_ADMIT_SAPS	1
☐	10	ICUSTAY_ADMIT_SOFA	1
☐	11	HR_ADMIT	1
☐	12	HR_DISCH	1
☐	13	WBC_DISCH	1
☐	14	HR_MAX	1
☐	15	TEMP_MAX	1
☐	16	WBC_MAX	1

Type: matrix
File Name: ICU.csv
Variable Name:
Dimensions: 15690x73
Type: matrix

Type: matrix
Variable Name: TrainTest
Type: matrix
Train Dimensions: 15376x73
Test Dimensions 314x73

Type: stat
Variable Name: Statistics
Column : 7: WEIGHT_ADMIT
Mean : 82.23
Median : 79.15
Mode : 70.00
Standard deviation : 22.41
Variance : 502.38
MAX : 300
MIN : 23

Figure 5.3 Block populated UI

· Taking up most of the screen is where action takes place. Is where blocks and elements are located. Users populate this area either by inputs in the browser window (buttons) or by sending requests to Alexa.

· To the left there are two lists. One stores all the available files which can be loaded at any moment and the other shows the current variables related to the last block if it has any. If not it saves the ones from the previous block. On small screens (lower than 1024px wide) the left bar is retracted into a menu icon and can be accessed that way.

· Finally, at the top a tool bar showing the title, B2SLab logo and a few buttons with not much use in this project but for resetting or loading files. Plus, the blue bar pulsates when an Alexa Session is active so that the user doesn't have to look at the Echo to know its listening.

Nearly every element has a number assigned. That is because Amazon doesn't encourage free form speaking to Alexa. It all relies in a series of slots which are predefined or can be customized. But in predefined slots voice recognition is significantly better so the easiest and handiest way to give inputs is to assign elements a number they can be called by. For example loading a file by saying "*Alexa, load file 4*" instead of trying to pronounce the file name.

5.2.2 Blocks

Each block has a content div to store either text or image and a group of buttons div. Text or image can't be modified once created the block.
First of the buttons is the one for numbering. It is disabled as it works as a block counter only. All blocks share it and it is impossible to have two blocks with the same number. Also every block has a delete button that can be pressed and deletes the block from the BlockChain. It can also be requested by voice to Alexa. Grey buttons show the options each block gives the user to continue the chain. This options are stored in a list of strings in each block. They can have functionality implemented or not but as the intended experience for users is to interact with Alexa it is discouraged. So its final utility is to prompt users with available actions to perform on the block.

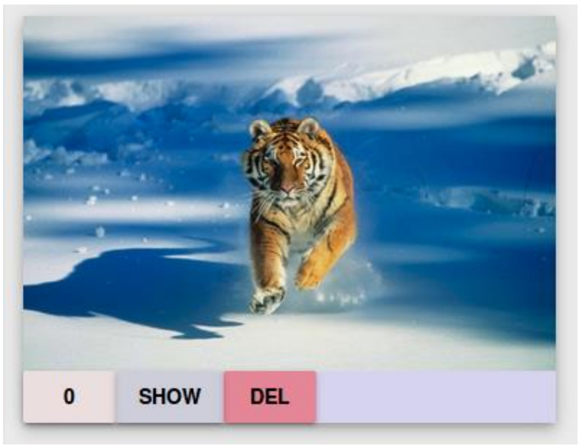
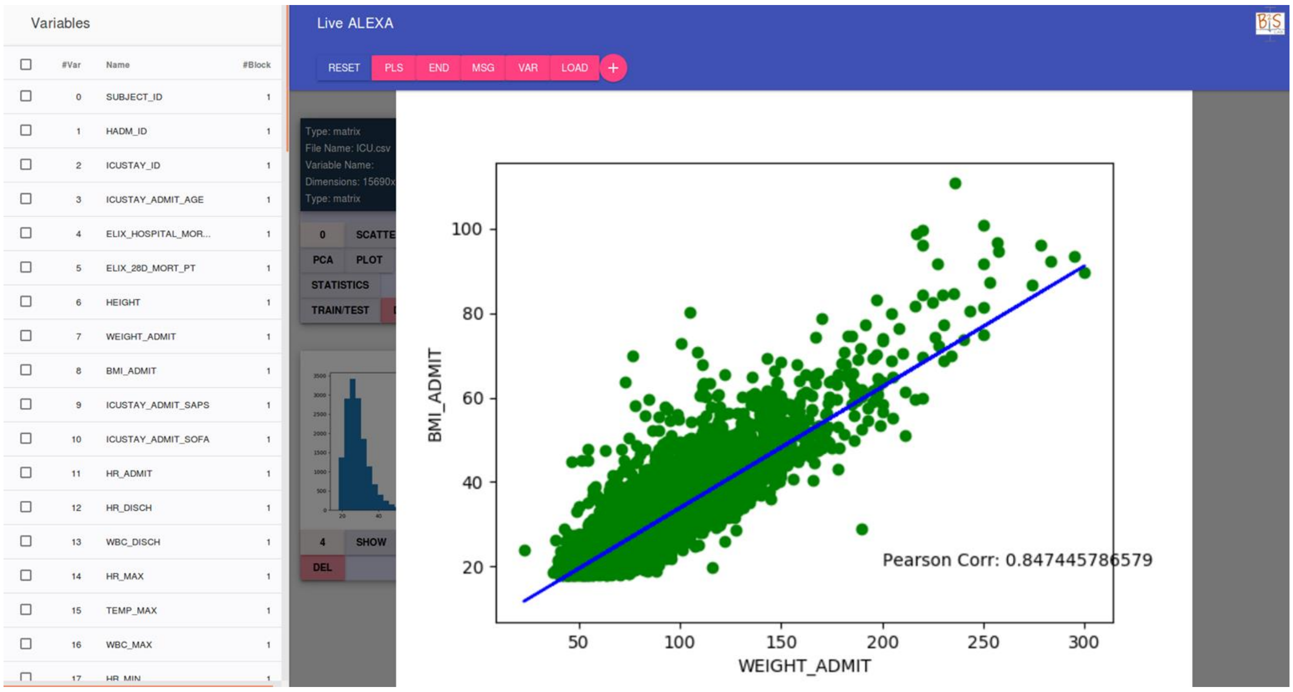


Figure 5.4 Image block example

A common button appears in every image block: show. It enlarges the image for a better view. It can be minimized by sending a request to Alexa or by clicking anywhere on the screen.



	#Var	Name	#Block
☐	0	SUBJECT_ID	1
☐	1	HADM_ID	1
☐	2	ICUSTAY_ID	1
☐	3	ICUSTAY_ADMIT_AGE	1
☐	4	ELIX_HOSPITAL_MOR...	1
☐	5	ELIX_28D_MORT_PT	1
☐	6	HEIGHT	1
☐	7	WEIGHT_ADMIT	1
☐	8	BMI_ADMIT	1
☐	9	ICUSTAY_ADMIT_SAPS	1
☐	10	ICUSTAY_ADMIT_SOFA	1
☐	11	HR_ADMIT	1
☐	12	HR_DISCH	1
☐	13	WBC_DISCH	1
☐	14	HR_MAX	1
☐	15	TEMP_MAX	1
☐	16	WBC_MAX	1
☐	17	HR_MIN	1

Type: matrix
File Name: ICU.csv
Variable Name:
Dimensions: 15690x
Type: matrix

Figure 5.5 showBlock() in action

5.2.2.1 IO_Block

The IO_Block displays some block information as the type or name. But the most important are the File from where it gets the data, ICU.csv, and the dimensions of the result matrix. Below the options that the user can perform from now on.

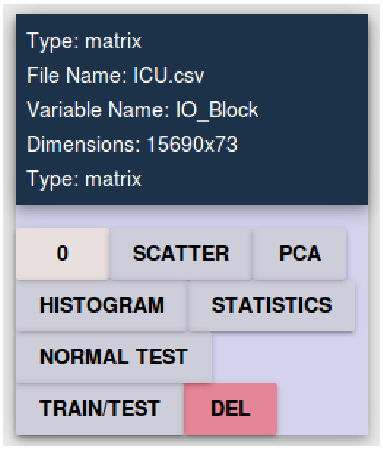

Type: matrix
File Name: ICU.csv
Variable Name: IO_Block
Dimensions: 15690x73
Type: matrix

Figure 5.6 IO_Block

5.2.2.2 TrainTestBlock

This block divides the initial matrix dimensions by separating it into two sets: train and test. It shows the adjustable dimensions for the two sets.

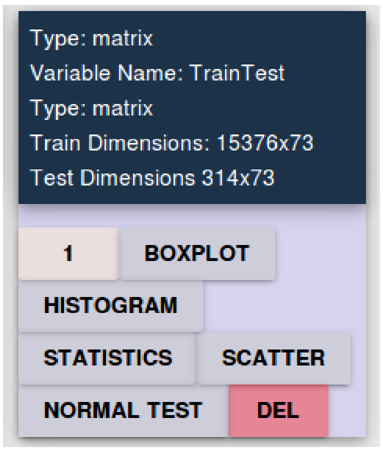
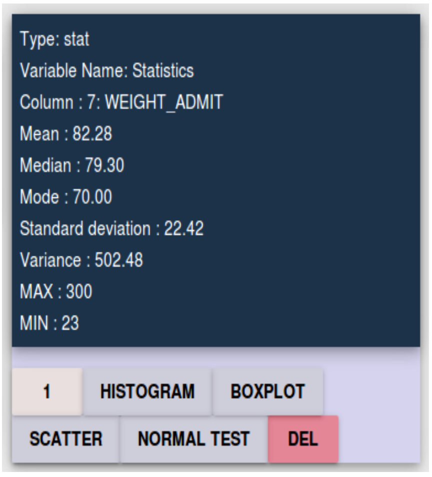

Type: matrix
Variable Name: TrainTest
Type: matrix
Train Dimensions: 15376x73
Test Dimensions 314x73

Figure 5.7 TrainTestBlock

5.2.2.3 StatisticsBlock

Shows all the basic statistics of a variable. It indicates the number and name of the variable.



```
Type: stat
Variable Name: Statistics
Column : 7: WEIGHT_ADMIT
Mean : 82.28
Median : 79.30
Mode : 70.00
Standard deviation : 22.42
Variance : 502.48
MAX : 300
MIN : 23
```

Figure 5.8 StatisticsBlock

5.2.2.4 HistogramBlock

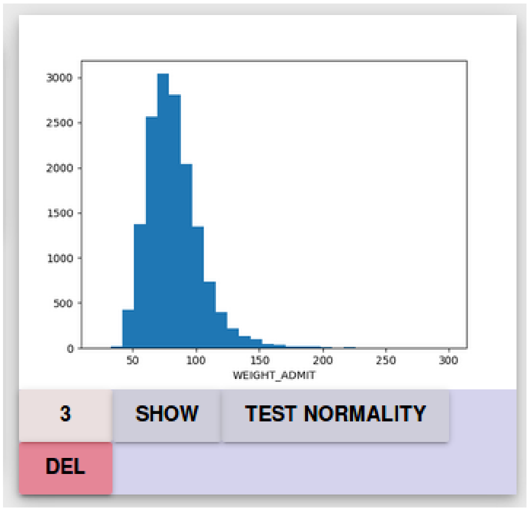


Figure 5.9 HistogramBlock

5.2.2.5 LinearRegressionBlock

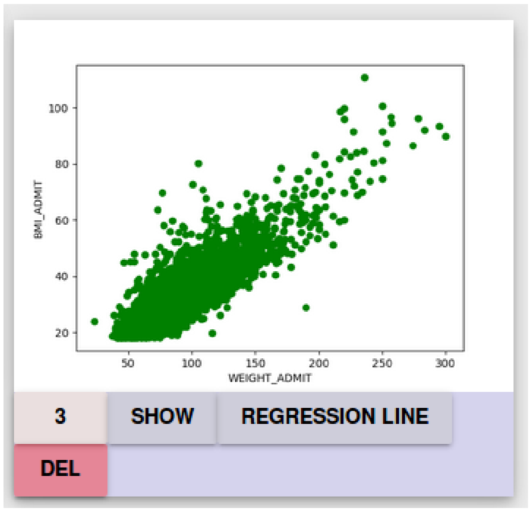


Figure 5.10. LinearRegressionblock scatter plot

Once called the regression line option with the passOption intent:

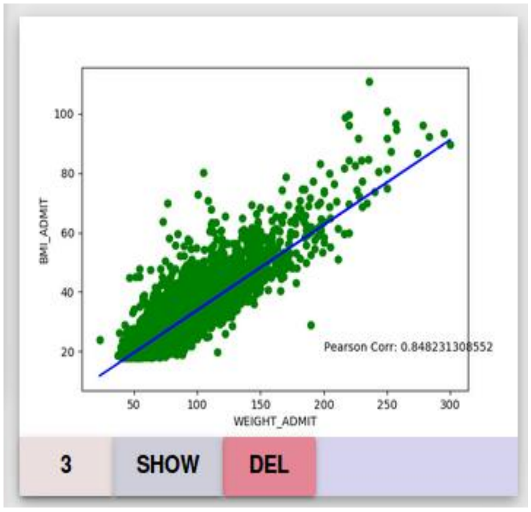


Figure 5.11 LinearRegressionBlock after passOption intent

5.2.2.6 BoxplotBlock

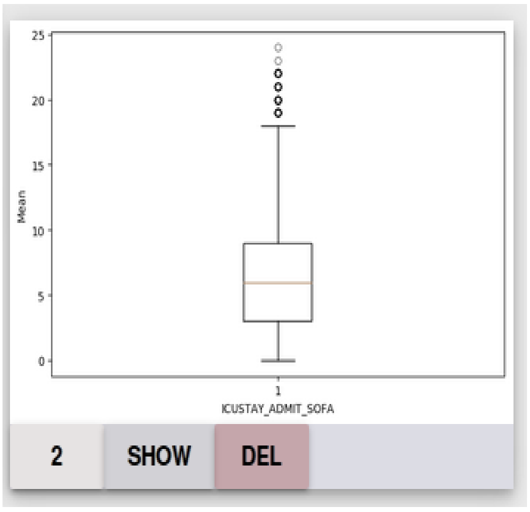


Figure 5.12 BoxplotBlock

5.3 Sequential organ failure assessment score (SOFA score)

5.3.1 Description of the score

Sequential organ failure assessment score (SOFA score), is used to track a person's status during the stay in an intensive care unit (ICU) to determine the extent of a person's organ function or rate of failure. The score is based on six different scores, one each for the respiratory, cardiovascular, hepatic, coagulation, renal and neurological systems. The score is also known as Sepsis-related organ failure assessment score, as it assists health care providers in estimating the risk of mortality due to sepsis. Sepsis is a life-threatening condition that arises when the body's response to infection causes injury to its own tissues and organs.

The patients variables related with the SOFA score are the following:

Respiration [PaO_2 / FiO_2 mmHg]

PaO_2 / FiO_2 mmHg	SOFA Score
<400	1
<300	2
<200	3
<100	4

Coagulation [Platelets x 10^3/µl]

Platelets x 10^3/µl	SOFA Score
<150	1
<100	2
<50	3
<20	4

Liver [Bilirubin mg/dl]

Bilirubin mg/dl	SOFA Score
1,2-1,9	1
2,0-5,9	2
6,0-11,9	3
>12,0	4

Cardiovascular [Hypotension]

Mean arterial pressure OR administration of vasopressors required	SOFA Score
MAP < 70 mm/Hg	1
Dopamine≤5µg/kg/min or Doboutamine (any)	2
dopamine > 5 µg/kg/min OR epinephrine ≤ 0.1 µg/kg/min OR norepinephrine ≤ 0.1 µg/kg/min	3
dopamine > 15 µg/kg/min OR epinephrine > 0.1 µg/kg/min OR norepinephrine > 0.1 µg/kg/min	4

CNS [Glasgow Coma Scale]

Glasgow coma scale	SOFA Score
13-14	1
10-12	2
6-9	3
<6	4

Renal [Creatinine mg/dl or Urine Output]

Creatinine mg/dl or Urine Output	SOFA Score
1.2–1.9	1
2.0–3.4	2
3.5–4.9	3
> 5.0	4

5.3.2 Prediction of the score

The prediction of the score is done by calculating the weight each parameter has in it and then building the best-fitting regression model. That will permit predicting future values from patients whose live is in danger.

The path it follows until the prediction is the following:

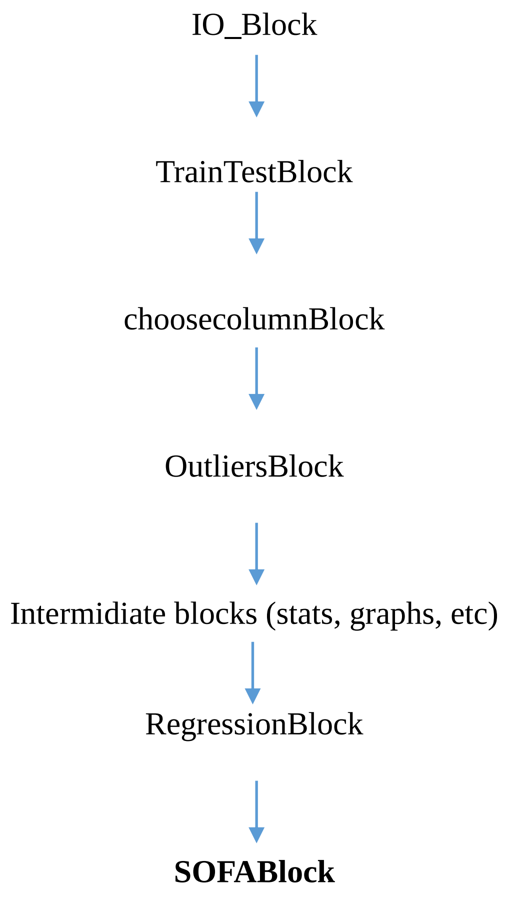


Figure 5.13: SOFA score block path

6. ECONOMICAL, ENVIRONMENTAL AND SOCIAL IMPACT

6.1 Economic analysis

This section details the estimated viability of the project from the developer's view (the student) as if it was to be sold for users.

During the fulfillment of the project there are some expenses to be considered:

· Student's work at an average 4 h/day during 5 months. If paid at 10€/h that equals 4.000€.

· The project director's work involved in 2h/week, which paid at 50€/h equals 2.000€.

· Personal computer bought the same year for the amount of 1000€.

· Amazon Echo device, which is valued in 149€ aprox.

If it was to be sold as a service for the amount of 10€/month, it would only take 60 people to use it to start producing benefits after one year. So it's clear it's a very viable project where money is not a problem. This project is thought for improving medical issues, so hospitals could have a discount as the main idea is helping and not profiting. Of course anyone who would like to use it will also have to buy the Echo device and the required equipment to make it work.

6.2 Environmental impact

The project's environmental impact is minimum. Office material and furniture is recyclable, whereas the personal computer and the cluster comply with the European regulation. We could study it's energy consumption and estimated negative emissions as CO_2 but, aside it's nearly negligible, it's only caused by office work which would have been done if not with this project with another.

6.3 Social impact

In terms of social impact, this project development does not directly create a negative impact on any society sector or group. What's more, it is thought to improve on a very important part of society with the only goal making things easier for medical staff, so it can only be looked by good eyes.
However, the data about the experiments may have a human precedence and follows a proper data protection policy. The datasets used in this project contain personal data that has been properly anonymized.

7. CONCLUSIONS

7.1 Results achieved

The results achieved are really satisfactory as all the implemented things work correctly. Even though there are some unfinished blocks, most of them show as expected and respond to Alexa the way it's supposed.

This project challenges on lots of problem solving issues which require big implication, heavy research and technical thinking.

Also, all the learning this project has supposed has been enormous. I've learned a whole new world with speech recognition and specifically in Amazon's Alexa. Also there's been great improvements on Python coding and introduction to Java programming.

7.2 Further work

This project was born with the idea of being a demonstrator of what could be done in a future. So there can be never-ending improvements on top of the work done which would make it much more powerful and even give way to a future useful tool. Forward programmers who enter this project have the background work already done for continuing building blocks and all it requires is people dreaming big and predicting future needs.

Further improvements examples could be:

· Creating a full patient comparison structure to ease medical staff work.

· Building a patient structure which gives all the possible treatments and previous scenarios where this treatments have proven useful.

· Identifying similar patient diseases and connecting them to find solutions for both.

This are some examples but expansion of this project can be infinite only delimited by people's imagination.

ACKNOLEDGEMENTS

To Alexandre, for giving me the opportunity of joining them in this project and helping when needed.

To Moritz. This project wouldn’t have been possible without his collaboration and teaching.

To UPC, for supporting the B2SLab.