Final Report

Creating An AI That Can Play a Tactical Intelligence Game Alongside Human Players

Robert Harris

C1334989

Supervisor: Professor Alun Preece

Moderator: Federico Cerutti

Module Code: CM3203

Module Title: One Semester Individual Project

Credits: 40

Project: 96

May 5th 2017

Abstract

The aim of this project is to determine whether a virtual system can be a viable asset in contributing to crowdsourced knowledge base. The project aims to answer this by implementing a system that can play the crowdsourcing knowledge game SHERLOCK. Using visual reasoning, linguistic reasoning and communication it is hoped the system can act as “one of the crowd” and contribute at the same level a human player would when playing the game. The project will use a mix of technologies such as; controlled natural languages, CENode and image recognition. The design and implementation will aim to have the attributes that are commonly found in a cognitive architecture.

The primary deliverable will see the AI player's contribution interact with the SHERLOCK dashboard alongside a human player's contribution. This contribution must then be “asked” for by a human player to determine that a human and virtual player can work together to produce crowdsourced knowledge.

Acknowledgments

I would like to acknowledge my supervisor Professor Alun Preece for his continued enthusiasm and guidance during the project's development. A further acknowledgement is to Dr Will Webberley whose guidance setting CENode and CEServer was invaluable in making this project a success.

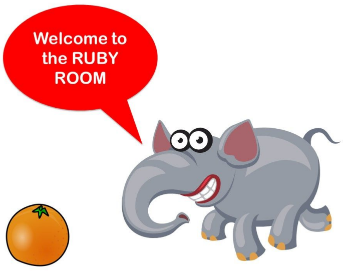
Finally I would like to acknowledge my housemate and friend Samuel Martin for his support and advice in implementing and writing the project.

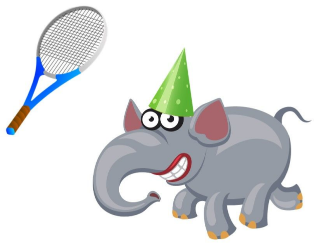
Table of Contents

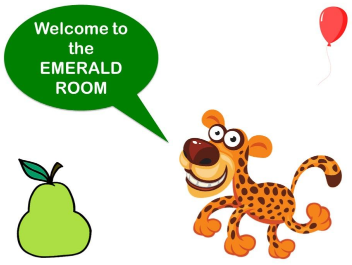
1.0 Introduction

The purpose of this project is to create an AI (Artificial Intelligence) which can play a crowdsourcing focused game alongside human players, and communicate what it sees to a central gossip agent. The game involves human players exploring a physical environment, finding posters and describing what they see in a natural language. To play alongside humans, the AI will receive visual and linguistic input, perform reasoning and produce a linguistic output. It will then communicate its linguistic output to a central agent similarly to human players. Systems that share similar components are traditionally known as cognitive architectures.

The tactical intelligence game at the forefront of this project is SHERLOCK (Simple Human Experiments Regarding Locally Observed Collective Knowledge) which traditionally involves human players performing intelligence, surveillance, and reconnaissance (ISR) tasks in a physical environment. A typical SHERLOCK game consists of Characters of Interest (CoI) displayed within poster scenes alongside accompanying attributes. Each poster will contain one character along with a differing number of the following attributes; location, types of fruit, types of sport, hat colour and varying objects [1]. The following images represents typical posters within the SHERLOCK game which humans will see.



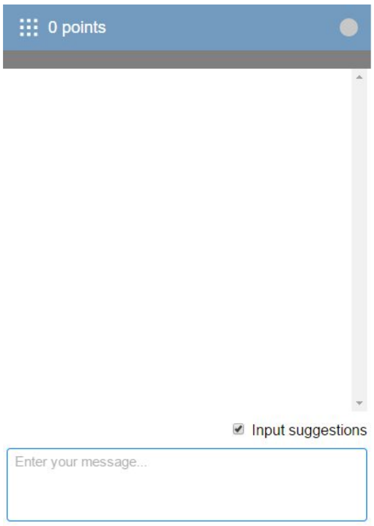




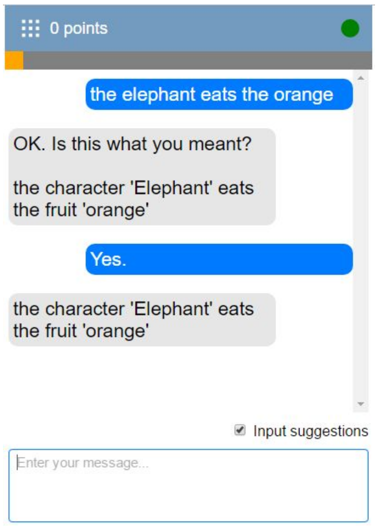
The first 2 images contain the elephant character with location information (Ruby Room), fruit (Orange), sport (Tennis) and hat colour (Green). The 3rd image contains the character Leopard with its location (Emerald Room), fruit (Pear) and an object (Balloon).

Operating as a team, human players are tasked with describing and reporting the scene objects they observe in natural language. Natural language observations can be reported using the SHERLOCK conversational interface, which in turn converts it into a controlled form of natural language to pass to the local user agent. This local agent will then update its local SHERLOCK dashboard accordingly. Interaction with the conversational interface can be seen below;

Conversational Interface



Natural Language Input



the elephant eats the orange

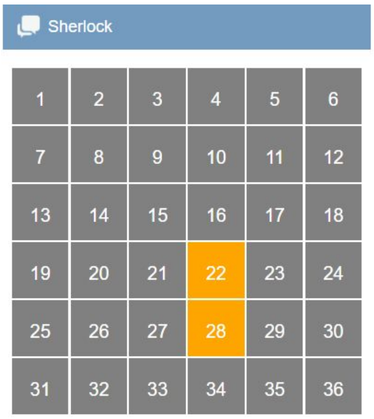
OK. Is this what you meant?

the character 'Elephant' eats the fruit 'orange'

Yes.

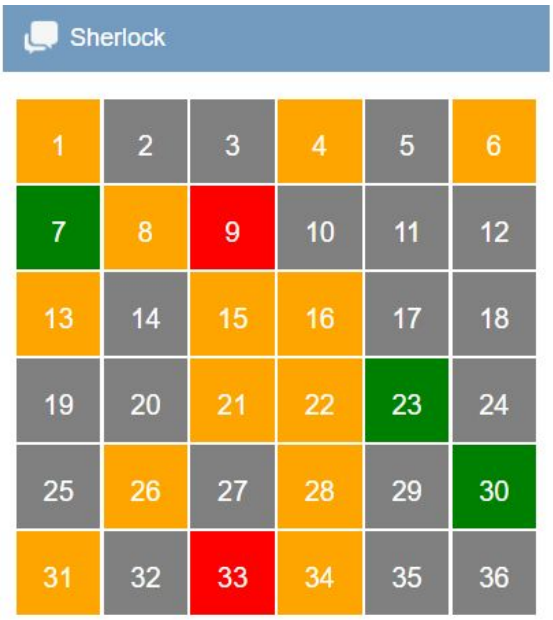
the character 'Elephant' eats the fruit 'orange'

Local SHERLOCK Dashboard



1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

The reported observations will initially update the user's local agent knowledge base, before enacting policies which relay its contents to a central “gossip” agent. This central agent will then distribute its collected knowledge to all players’ local agent, updating their SHERLOCK dashboard appropriately. A dashboard representing the combination of observations can be seen below;

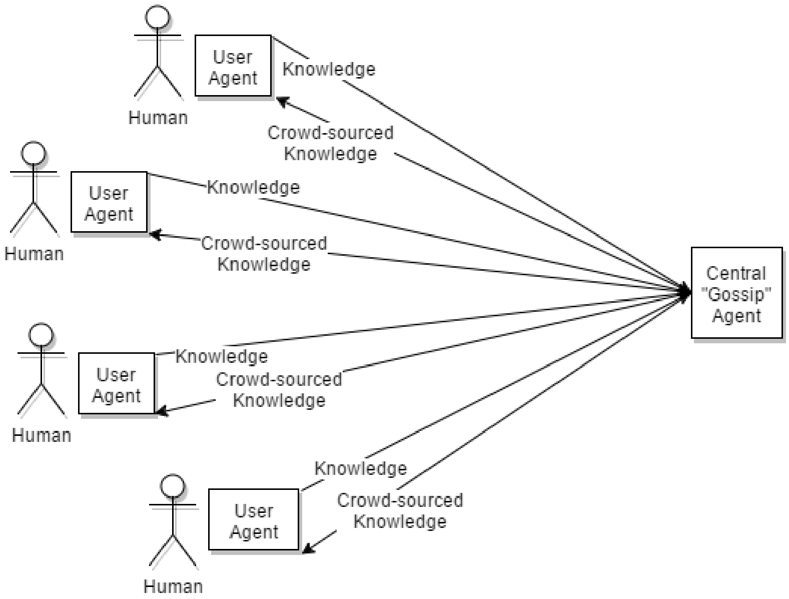


1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

There are 4 different colours represented by the board; green implies that players have come to the same conclusion, red implies that there are contradictory statements regarding the same question, yellow implies some information is available but not enough to draw a conclusions, and grey implies no information has been received regarding that question.

The central agent will distribute its knowledge at regular intervals to ensure all agents have up to date knowledge about the environment.

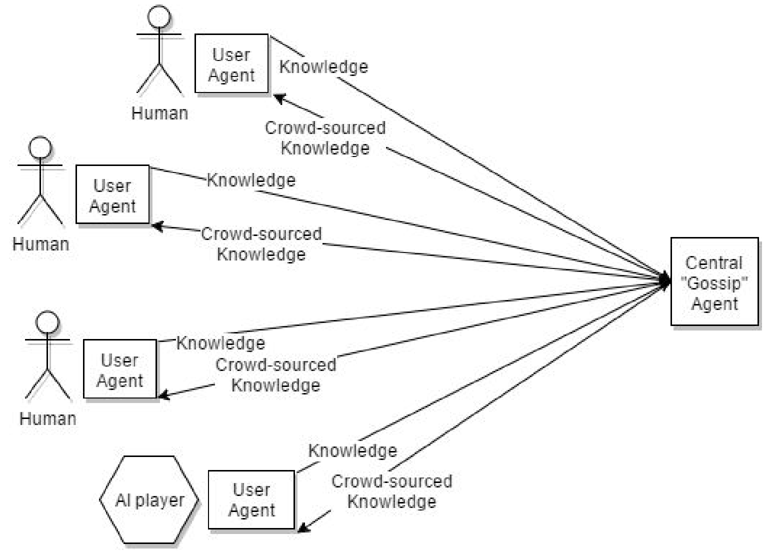
The following diagram attempts to illustrate how a player's local user agent will communicate with the central agent.



A user's agent will relay its knowledge as well as receive crowdsourced knowledge from the central agent.

The aim of the AI player is to be equal to the human player in terms of capability, and to be one of the crowd amongst the agents playing the game. To be an equal to the human player it will need to perceive, describe and communicate observations it makes to the central agent. The AI players observations should then be visible and queryable by human players. Due to the absence of a physical body, the AI player will need to perceive posters through feeds rather than exploring a physical environment. The AI player should also have its own local agent to maintain, so it can relay and receive knowledge from the central agent.

A game environment containing the AI player can be illustrated with the following diagram.



The above diagram illustrates how the AI player should play the game in a similar manner to a human player.

In this project I'll discuss how I've attempted to model my high level design and implementation on cognitive architectures. Taking influence from a cognitive architecture I can attempt to modularise human behaviour in the program (eg: vision, language, communication).

Overall, this project aims to prove the viability of an AI player in contributing to a crowdsourcing activity by combining computer vision, reasoning and communication techniques. Proving the technique works in a controlled environment opens it up to be further expanded into real life situations. For example, a human's ability to perceive, describe and adapt to an environment is superior to a computer's, however a computer has the ability to remotely operate in inaccessible or potentially dangerous environments to humans. In a future project, we could aim to combine the strengths of both humans and computers while performing crowdsourcing activities.

The report structure is as follows: Chapter 2 introduces the related work that has guided this project, including cognitive architectures, controlled natural languages and computer vision.

Chapter 3 will focus on the design and architecture of the software, including system scope, requirements, use cases and boundaries.

Chapter 4 will contain an in-depth description and discussion about the software implementation including code snippets and justifications.

Chapter 5 will focus on testing and the techniques used to ensure different systems were communicating with each other.

Chapter 6 will focus on the potential for future work, discussing how the techniques used can be expanded and re-applied in situations where crowdsourcing knowledge and creating shared awareness is a key goal.

Chapter 7 will reflect on the overall project including design choices, implementation and underlying assumptions.

Chapter 8 will conclude the project and summarise my findings.

During development and testing, no personally identifiable or compromising information has been collected or processed by myself or the system.

2.0 Background

To design and implement an AI which is modeled on human behaviour, a deep understanding was required in several key areas. These areas include cognitive architectures, computer vision techniques and controlled natural language research. In this chapter I will thoroughly explore each area and expand on how each topic relates to the overall objective.

2.1 Cognitive Architecture

Paul Rosenbloom from the Institute for Creative Technologies defines a cognitive architecture as a "*hypothesis about: (1) the fixed structures that provide a mind, whether in natural or artificial systems; and (2) how they work together – in conjunction with knowledge and skills embodied within the architecture – to yield intelligent behavior in a diversity of complex environments.*"[4]

In its most most basic form, a cognitive architecture specifies the underlying infrastructure for an intelligent system. Focusing on the theoretical structure of the human mind, cognitive architectures aim to summarise results of cognitive psychology into an extensive computer model. An architecture would include those aspects of a cognitive agent that are constant over time and across different application domains. Just as different programs can be run on the same computer architecture, different knowledge bases and beliefs can be interpreted by the same cognitive architecture.

Langley, Laird and Rogers [3] state a cognitive agent would include the following capabilities: short-term and long-term memories that store the agent's beliefs, goals and knowledge; the representation of elements that are contained in these memories and their organization into larger-scale mental structures; and the functional processes that operate on these structures, including the performance mechanisms that utilize them and the learning mechanisms that alter them.

Traditionally cognitive architectures can be symbolic, connectionist or hybrid. Symbolic cognitive architectures tend to be based on generic rules or the mind-is-like a computer analogy (SOAR [5]). In contrast, the sub-symbolic nature of a connectionist cognitive architecture specifies no such rules based on previous examination, and relies on emergent properties of processing units (CLARION [6]). The hybrid approach combines both symbolic and connectionist processing.

2.11 ACT-R Cognitive Architecture

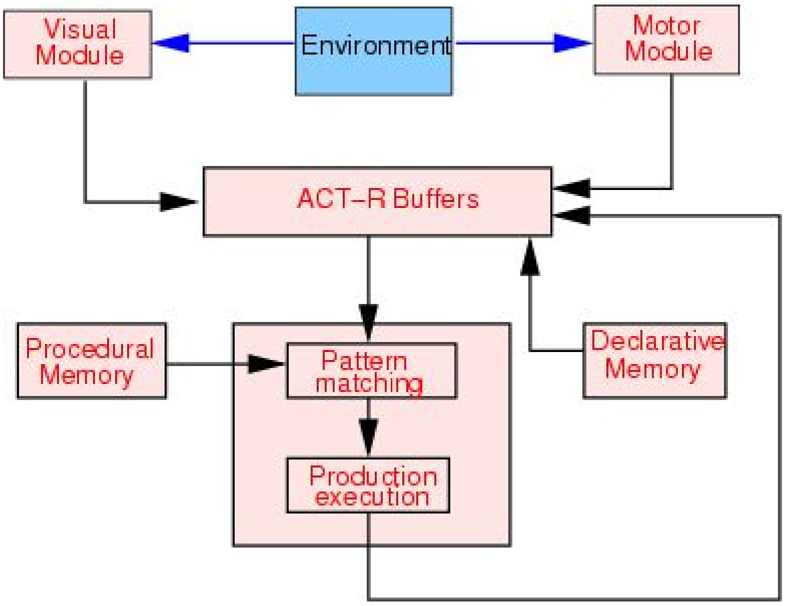
With the aim of harnessing cognitive abilities such as vision, reasoning and linguistic communication, it was important I attempt to model my design off an already existing cognitive architecture. The architecture I have chosen is ACT-R (Adaptive Control of Thought - Rational) which was developed by John Robert Anderson at the Carnegie Mellon University. ACT-R was chosen due to its already established role in modelling and solving problems in a similar fashion to the human mind.

ACT-R is an example of a hybrid approach to cognitive architecture processing, as its symbolic structure is a production system, whereas the subsymbolic structure is represented by a set of massively parallel processes that can be summarized by mathematical equations. These subsymbolic equations control many of the symbolic processes [2].

ACT-R's approach to modelling human cognition can, on the exterior look like a programming language; however it's constructs reflect assumptions about human cognition, derived from numerous psychological experiments. In a similar vein to a programming language, ACT-R is a framework for different tasks e.g (language comprehension, communication, memory for texts/ words). Researchers develop models written in ACT-R, that incorporate ACT-R's view of cognition as well as their own assumptions for that specific task. A distinguishable feature of ACT-R over other cognitive architectures, is its ability to collect quantitative measures that can be directly compared with that of human. For example, a researcher can directly compare the time to perform a task and the accuracy of a task for a human and an ACT-R model directly. This feature will be applicable to my application as the AI player will be performing the same task alongside humans.

2.12 How ACT-R Works

The ACT-R cognitive architecture contains 3 main components; a pattern matcher, modules and buffers. The following diagram illustrates how the different components of a traditional ACT-R model interact.



[7]

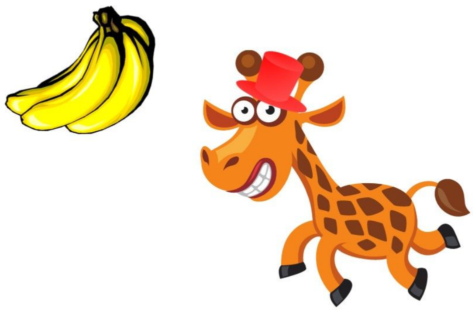
The 2 main types of modules in ACT-R include perceptual-motor modules and memory modules. Perceptual-motor modules supply an interface to the real world environment, with most the popular perceptual-motor modules including visual and manual. Memory modules consist of declarative memory (long term memory) which store facts, and procedural memory which are made of production rules (if-then rule model). ACT-R accesses its modules through buffers, with each module having a dedicated buffer to serve as the interface to that module. The buffers in a sense act as short-term memory, with their state being modified regularly. The pattern matching component searches for productions (rules) that match the current state of the buffers. Only one production can be executed at a single time. When executed, the production can modify the buffers and change the state of the system, thus cognition unfolds as a succession of production firings.

The goal of my implementation was to take inspiration from the modelling paradigms of ACT-R and design my system in a similar fashion. I aim to have 3 perceptual motor modules consisting of a linguistic module and visual module and communication module. This is followed by the declarative memory which will be in the form of our local user agent. The production rules, pattern matching and execution will be performed by the code itself. The modules will aim to act as an interface between the environment, whereas the buffers act as the interface between the modules and production rules, pattern matching and production execution. The buffers in my context will be the content of variables passed between modules. The environment will be other human players' observations, poster scenes (visual input) and human interaction with the AI player (question input).

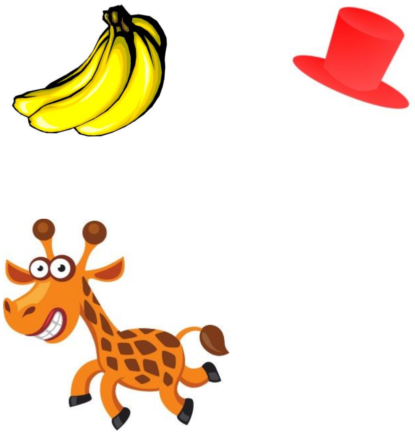
2.2 Computer Vision

The AI player will use computer vision to perceive the environment and determine which objects are present in it. Given the below poster scene, computer vision will be used to identify the objects within it.

Scene



Items to Find



Our computer vision implementation should be able to find all items within scenes, and when prompted, return the item that's been requested (eg: if we're looking for which fruit the Giraffe eats then we'd return the fruit banana). As we are only finding simple 2D objects in simple 2D scenes, we can use a well performing off the shelf solution which can achieve this functionality. As my implementation is not in real time, the engine can be located on a remote system and not on a handheld device. This design choice has allowed me to explore several types of descriptors regardless of resource usage, before deciding on a final one.

The vision module defined in ACT-R is where visual input is received from the environment in the form of pixels. The goal of this module is to mimic human vision which is sensed through the eyes and processed by the brain. Computer vision is the science that attempts to replicate this function by giving a machine similar or better visual sensing and processing. It is primarily focused on the automatic extraction, analysis and understanding of useful information from a single or sequence of images. There are numerous areas where computer vision is applicable, with a subset including; augmented reality, facial recognition, forensics, remote sensing, robotics, security and surveillance [8].

2.21 How Image Recognition is Currently Achieved

Image recognition requires 3 key components in order to function. The first being a set of training images to train the descriptors, the second being the descriptor used to gain information from an image, and the third being the matcher which is how training images are matched with query images.

Training images are an critical component, as these images are what query images attempt to match too when the algorithm is run. They can take the form of a single image or a set of images, which will train the detector in finding images of a similar nature. In the context of this project, our training images will be the individual objects within the game (Appendix 7), and our query images will be the scene posters featured in the game (Appendix 8).

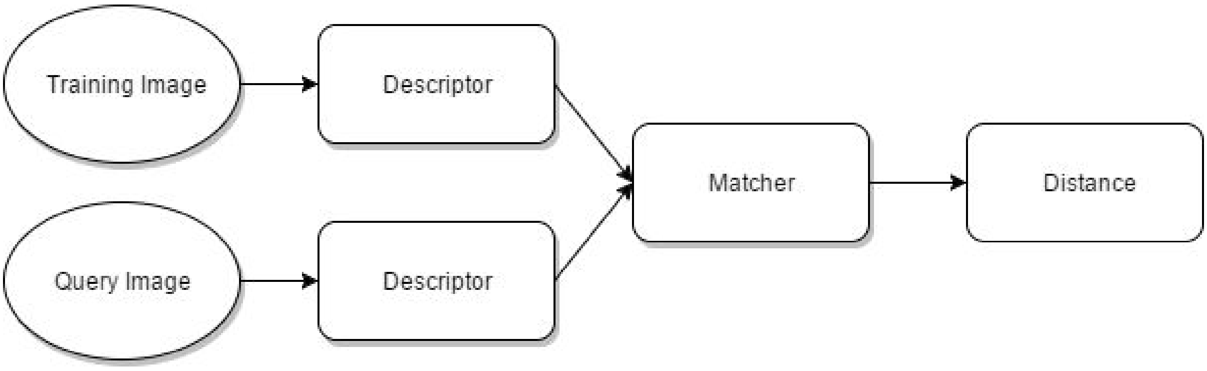
To identify areas of interest in an image, feature descriptors are used. The descriptor component classifies and stores information about pixel areas of a training image, and uses them to recognise similar pixels in query images. The two most widely used descriptors are texture descriptors and key-point descriptors.

Given a texture area, texture descriptors uniformly process the entire area and extract a high number of parameters to describe it. These parameters tend to be very similar or identical for each patch of the input area. These parameters tend to be of low quality by themselves, however in high numbers can be useful for a classification task.

Key points on the other hand can be defined as spatial locations or points in the image that stand out due to their interesting nature or “uniqueness”. Key-point descriptors only process pixels that have a high enough uniqueness value, meaning each pixel will be given a uniqueness measurement before being processed. The uniqueness value measures how recognisable that pixel will be in another image, with the top rated pixels being processed into descriptor objects. Keypoint detection is scale invariant, meaning it's unaffected by changes in image rotation, shrinking, expanding and distortion. Being scale invariant, one should be able to detect the same key points in a modified image as one can in the original.

Both techniques will invoke descriptor methods which produce descriptor objects. Formed from training images, key point descriptors contain such data as; scale, orientation, edges, gradients and corners relating to key points. Texture descriptors on the other hand contain data relating to features, templates and image segments extracted from texture areas.

The final component to image classification is the matching process. The matcher is responsible for matching a query image with training images. This'll determine whether an object exists or not. This component is very much dependent on the ability of the descriptor to identify areas of interest, meaning it's important the correct image classification technique is used for the problem.



The above diagram illustrates the typical flow of using a key point descriptor. It would first take a training and query image, and describe them with a descriptor. These 2 descriptors can then be compared against each other with a matcher, producing a distance parameter to indicate similarity.

2.22 Choosing the Descriptor

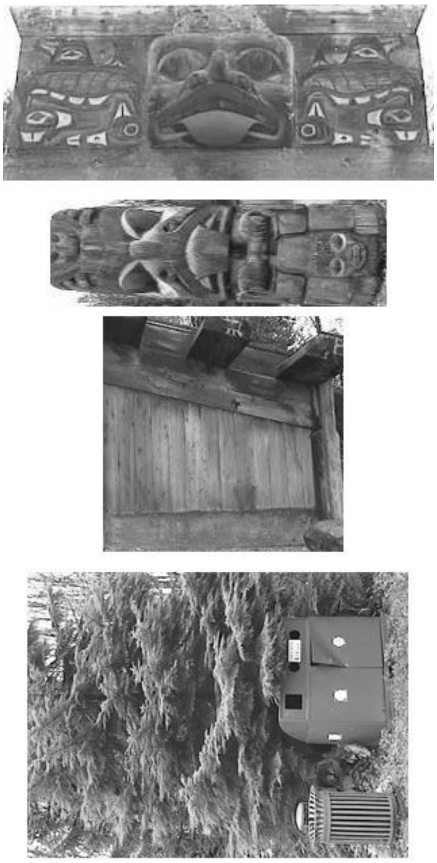
The descriptor was chosen early on in the project lifecycle and involved research and experimentation of both texture and key point descriptors. Firstly I experimented with the texture descriptor Histogram of Gradients, however I decided early on that I would not be implementing a texture descriptor for my visual module. This is because the Histogram of Gradients descriptor along with other similar texture descriptors (Local Binary Patterns, Haar), require thousands of positive and negative images stored in a classifier to perform well. This project would require thousands of positive and negative images of each SHERLOCK object, which would take a lot of time to compile (the images) and to train each descriptor.

The solution therefore was to use a key point detector as it only needs a single image to be trained. The SIFT (Scale Invariant Feature Transform) descriptor was chosen over other key point descriptors due to its high accuracy rate, scale invariance and open source availability. Further advantages included; its locality, meaning its features are local and robust to occlusion and clutter. It’s distinctiveness, allowing individual features to be matched in a large database of objects. It’s quantity and efficiency allowing many features to be generated for even small objects, while being close to real time performance. Finally, its extensibility means it can be easily extended to a wide range of differing feature types, with each adding robustness [19].

2.23 The SIFT algorithm

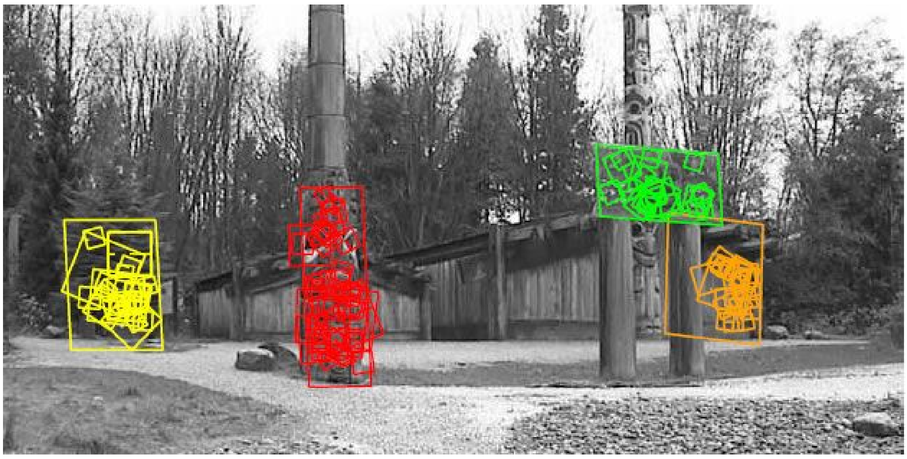
The SIFT algorithm is one of the most well established in the field of computer vision and in recent times has been noted as the “classic” approach to image recognition problems. Although SIFT is scale invariant, it can also handle changes in rotation, illumination and viewpoint with good results.

Given the following set of training images;



and the following scene;

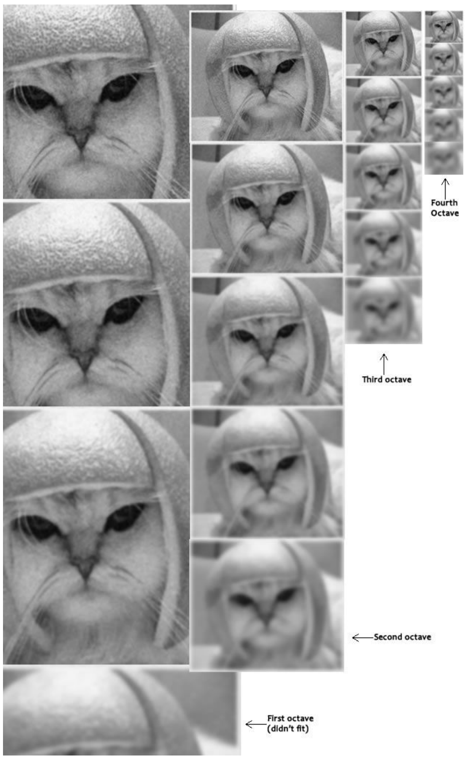
our SIFT algorithm will match to the following points.



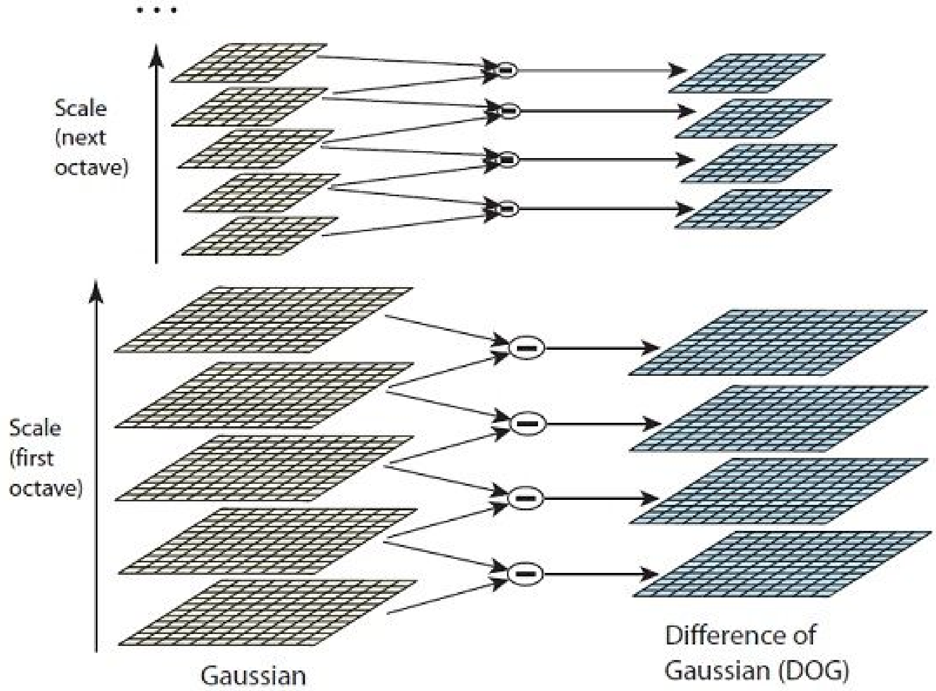
The large rectangles mark the matched images, with the smaller squares denoting individual features in those regions[9].

2.231 How it Works

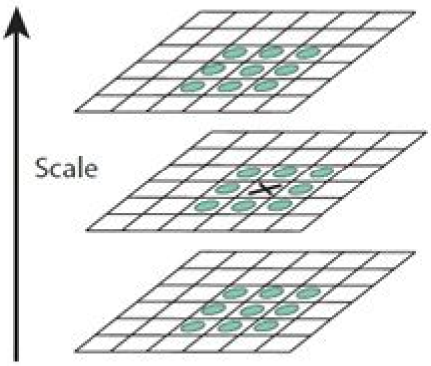
The first step in the SIFT algorithm is to ensure scale invariance. This can be achieved by creating an internal representation of the original image by generating a scale space. Creating a scale space in SIFT involves taking the original image and iteratively applying a Gaussian blur to it. This will form an octave of images. The original image will then need to be resized by half, followed by an iterative blur of the resized image [10]. This process is repeated on smaller variations of the image, producing several octave layers. The following image provides a visualisation of the process.



The next step is to use the blurred images produced earlier and generate a new set of images, known as the Difference of Gaussians (DoG). These DoG images are vital for detecting blobs at different scales, however can be computationally expensive. To address we use scale space to calculate the difference between two consecutive scales. This process is repeated for all octaves. The following image illustrates this.



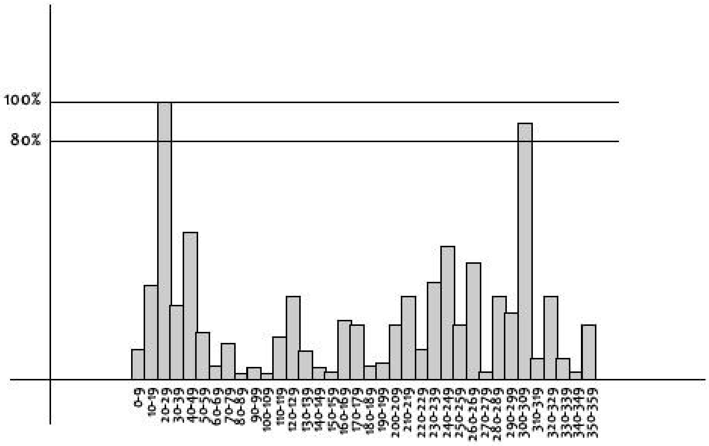
These DoG images are approximately equivalent to the Laplacian of Gaussian, with the added benefit of being scale invariant. Although being an approximation, there are no negative effects to the accuracy of the algorithm [11]. The next step is to find the key points in the image. This can be done by detecting the maxima and minima of the DoG images by comparing neighbouring pixels in the current scale, the scale above and the scale below. Assuming X is a key point, the diagram illustrates how the algorithm will compare with its neighbours.



This is followed by mathematically locating the subpixel maxima and minima by applying Taylor expansion to the approximate keypoint [12]. On solving we'll have subpixel key point locations, which increase the chances of matching, and improve the stability of the algorithm [13].

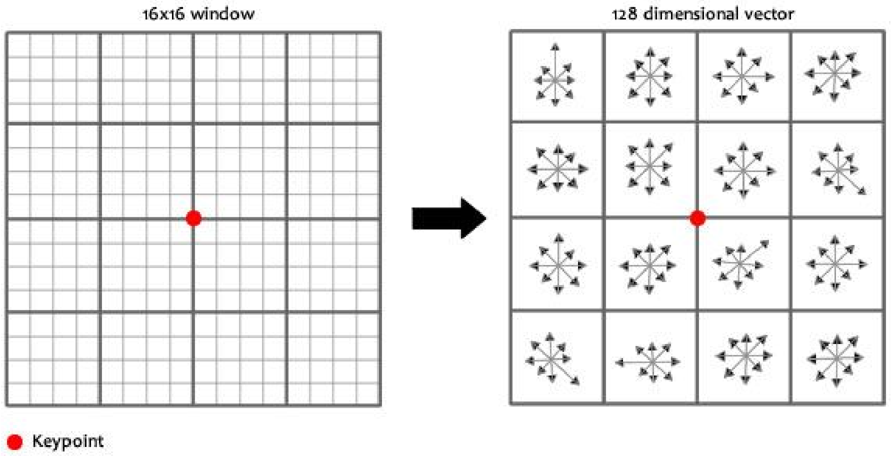
The next step is to dispose of low contrast keypoints generated in the previous step. We can achieve this by reapplying the taylor expansion to get the intensity value at the subpixel locations. If it's a magnitude less that a certain value the keypoint is rejected. Keypoints are generally rejected if they had a low contrast or are located on an edge. The process of rejection helps increase the efficiency and robustness of the algorithm [14].

Left with a set of legitimate key points, the next phase is to assign an orientation to each point. This orientation will provide rotation invariance to the detector. This process involves the collection of gradient directions and magnitudes around each keypoint, followed by the calculation of the most prominent orientations in the region. These orientations are then assigned to the keypoint, ensuring rotation invariance. Generally a histogram is used to identify the most prominent gradient orientations, with a peak of over 80% marking a new keypoint. If there are multiple peaks above the 80% mark then they're all converted to a new keypoint in respect to their orientations. The following diagram illustrates this;

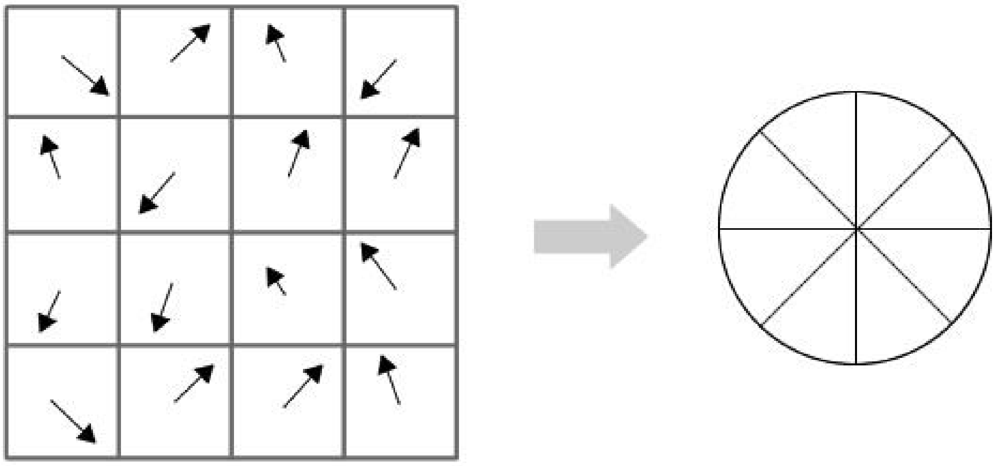


[15]

The final step is to generate the feature by creating a fingerprint for each keypoint. This will allow us to distinguish keypoints from each other. Given the human face, one keypoint may represent an ear, one may represent a nose while another may represent the eyes. To create a fingerprint, one must first take a 16x16 window of “in-between” pixels around the keypoint, and split that window into sixteen 4x4 windows.



From each 4x4 window you generate a histogram of 8 bins. Each bin corresponds to 0-44, 45-89, … , 315 - 359 degrees. Gradient orientations from the 4x4 windows are put into these bins, which is then repeated for all 4x4 blocks.



Finally the 128 values you get are normalised to form a 128-dimensional feature vector, based on the gradient orientations of pixels in 16 local neighbourhoods [16].

2.23 Open Source Computer Vision Library (OpenCV)

Image processing will be achieved using the open source vision library OpenCV [17]. Released under a BSD license, OpenCV is free to use for both academic and commercial use. OpenCV would be ideal for this project as it has support for classification techniques such as SIFT, SERF and MESR, along with compatibility with all major operating systems. Although OpenCV is implemented in C/C++, it has support for Python and Java wrappers allowing me to implement a Python or Java image classification system. As the wrappers will be executing the native C/C++ code, there will be no adverse performance calling them from a wrapper. The SIFT algorithm is patented for commercial use in the USA [18], however licensing should not be an issue as it’s free to use for academia, coupled with the lack of jurisdiction of US patents in EU states.

2.3 Natural Language Processing

The AI player does not need complex natural language processing capabilities, as its exclusively designed to play the SHERLOCK game. As its not designed to be a general AI, we can limit its natural language processing to the level of that in the game itself. As the game uses Controlled English to describe its environment, we will also produce CE to describe the environment. Using an already specified controlled natural english framework along with an established processing environment for CE, my AI players can communicate its observations with other players in the SHERLOCK game. Using the same controlled natural language framework as the game itself, compatibility has been easily achievable with the AI player.

2.31 Controlled Natural Languages

A Controlled Natural Language (CNL) is a form of Human Computer Collaboration

(HCC) focusing on ways humans can best communicate with a machine. The two traditional channels of HCC involve giving computers human-like abilities, generally focusing on language; with the second being a machine's ability to work alongside humans. In Loren G Terveen's paper, 'Overview of Human-Computer Collaboration', he suggests a combination of both approaches is the most effective way of encouraging human-computer collaboration [20].

The combined approach to HCC has been widely accepted by computer scientists, social scientists and linguists, resulting in the development of technologies to complement it. This view however presents a challenge in regards to communication between humans and machines. Humans prefer communication via natural languages and images, whereas machines prefer communication in a strict and controlled form. While machines can process natural language and images to some extent, without any form of control it can lead to ambiguity and miscommunication.

A CNL takes onboard the unified research findings of HCC, and provides a compromise between both humans machines. A CNL is viewed a subset of a natural language as it introduces restrictions on vocabulary. These restrictions aim for it to be easily processed by a machine, while also being human-readable and writeable.

In the 2014 paper 'A Survey and Classification of Controlled Natural Languages', [21] Tobias Kuhn attempts to bring order to the variety of english CNLs, by presenting a general classification scheme. Kuhn provides a comprehensive survey of existing English based CNLs, listing and describing 100 languages dating from the 1930's to the present day.

Kuhn explains how CNLs have originated from multiple disciplines (computer science, philosophy, linguistics, and engineering) and spanned over many decades (from the 1930's), with each background continuing to use a different name for the same kind of language. He adds that many CNLs share similar properties, however exhibit a wide variety of difference: some CNLs are ambiguous, others are as precise as formal logic; virtually everything can be expressed in some, only very little in others; some look perfectly natural, others look more like programming languages; some are defined by just a handful of grammar rules, others are so complex that no complete grammar exists.

Kuhn discusses how past attempts at defining a common core language have been inconclusive, therefore he proposes a definition. Kuhn states;

A language is called a controlled natural language if and only if it has all of the following four properties:

1. It is based on exactly one natural language (its "base language").

2. The most important difference between it and its base language (but not necessarily the only one) is that it is more restrictive concerning lexicon, syntax, and/or semantics.

3. It preserves most of the natural properties of its base language, so that *speakers of the base language can intuitively and correctly understand texts in the controlled natural language, at least to a substantial degree.*

4. It is a constructed language, which means that it is explicitly and consciously defined, and is not the product of an implicit and natural process (even though it is based on a natural language that is the product of an implicit and natural process).

A further shorter definition was also provided;

A controlled natural language is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics, while preserving most of its natural properties.

One of the most prominent CNLs was developed by IBM, and is defined by the International Technology Alliance as a form of Controlled English (CE) [22]. Although inspired by CLCE (Common Logic Controlled English), ITA CE is less strict in terms of precision: It has an “Informal meaning and a semi-formal mapping to predicate logic”. The two most common forms of language rules are “logical rules” and “rationale” statements. A “logical rule” would look similar to the following: “if (the person X has the person Y as brother) and (the person Z has the person X as father) then (the person Z has the person Y as uncle) ”. A “rationale” rule may look like: “(“the plan has failed” because “there was a misunderstanding”.)” [23].

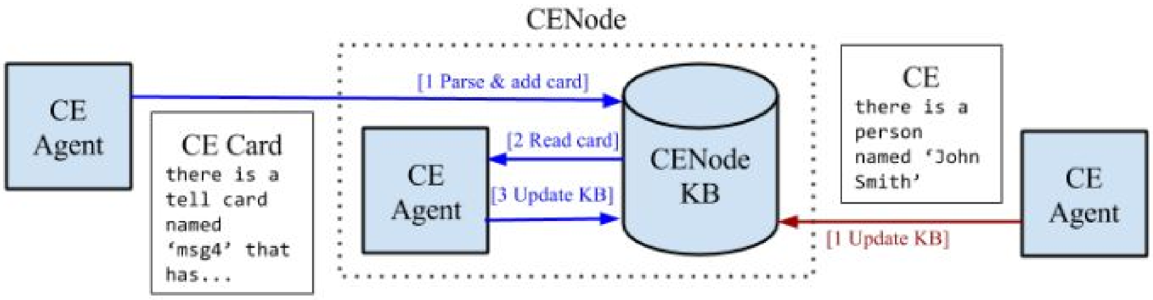
Using ITA CE, the AI player can produce should be able to produce output in the following format;

```
The hat colour 'green' is worn by the character 'Elephant'
The sport 'basketball' is played by the character 'Leopard'
The fruit 'banana' is eaten by the character 'Giraffe'
The character 'Lion' is in the room 'Amber Room'
```

2.32 Controlled English Node (CENode)

CENode is a JavaScript implementation of the ITA project's CEStore system [24]. It comprises of a knowledge base capable of modelling entities and their relationships, and a central agent that enables humans and other agents to update and query this KB directly using ITA CE (Controlled English). CENode instances are capable of being deployed in a variety of settings, such as web applications, standalone apps on handheld devices and servers [25].

The following diagram illustrates how one could manipulate the knowledge base directly or through cards.



In Will Webberley's paper "CENode: Enabling Human-Machine Conversations at the Network Edge" [26], he gives an in depth description on how knowledge base manipulation takes place.

Each CENode comprises a KB and a local CE agent that maintains the KB, shown above. A CENode will try to process and update its KB when any CE is received. As with the CE Store, CENode instances also support the blackboard architecture, which enables users and agents to submit CE sentences wrapped in CE Cards that are addressed to the local agent. If a card addressed to the agent is received, then the agent can find the card and read it. If the card contains valid CE, then the agent can then use this to modify its KB. If the card is not addressed to the local agent, then it will remain in the KB unread.

Webberley continues by providing an example interaction with the KB via CE;

Assuming a node's local agent is named agent1, the following two sentences received by the node would have equal effect:
'there is a tell card named 'msg1' that is to the agent 'agent1' and has 'there is a subject named Computing' as content.'
'there is a subject named Computing.'

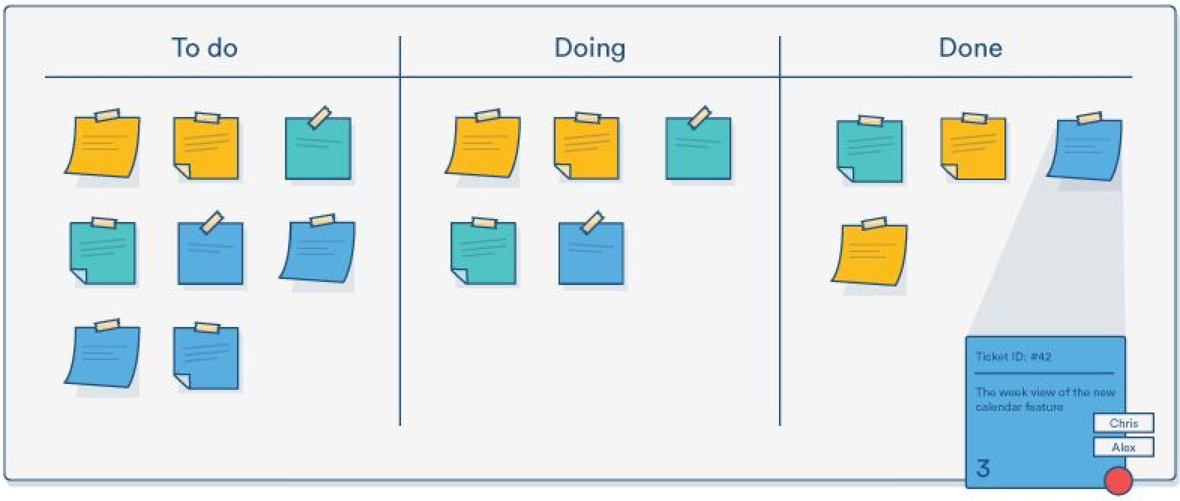
The AI player will take advantage of the rich API provided by CENode to host a CE local model of the SHERLOCK game. Using a GET call, the AI player can query a localhost model for information and resources regarding the game environment. Using the POST call, the AI player can upload a card of their described scene in CE to the central agent. On a local scale, CENode provides the architecture for interacting with the declarative memory, by maintaining its own KB as well as being told other players' observations. In the context of the AI player, CENode will be interacted with by the the communication module which shares data with the central agent and queries data from its local agent. This sharing of data is only possible due to CENode's automatic interaction capability with other CEnode nodes on the network. Using CE policies CENode can inform a node to tell all other CENde nodes (via HTTP) when information is added to the knowledge base.

3.0 Approach and System Design

In this section I shall give an overview into my approach while developing the AI node, as well as discussing the system design and decisions associated with it.

3.1 Approach

The development approach deployed on this project borrowed heavily from the agile software development methodology. This style was chosen due to its ability to succeed in a turbulent and changing environment. Further advantages included its emphasis on developer/ stakeholder interaction, which fit perfectly into the student/ supervisor dynamic. This close interaction allowed for the rapid implementation of changes and the swift addressing of problems during development. With supervisor meetings varying from weekly to fortnightly, I could model my development sprints between the meeting dates, presenting a deliverable each time. Smaller changes could still be made during sprints via interaction on the real-time messaging service Slack [28]. Aspects of the agile approach that were not deployed were scrums, as they were not an applicable tool in an individual project. In its place, I utilised Kanban which is a method for managing knowledge work. Kanban allowed me to balance demands for work with the available capacity for new work. I found this method of organisation a good complement to the agile style of development, due to its use of visualisation to monitor workflow and set development targets. The following is a traditional example of the kanban method work management system.



Over the course of the project I utilised 4 sprints, each with its own goals, objectives and deliverables. Following is an overview of the 4 sprints;

Sprint 1

The first sprint focused on my initial research and implementation of computer vision techniques. Using the OpenCV vision library, I experimented with the histogram of gradients texture descriptor, along with the ORB and SIFT keypoint detectors. Experimentation with each descriptor involved the detection of an easily identifiable image within a basic non cluttered scene. It was during this sprint that the SIFT detector was settled on for image recognition. The sprint deliverable focused on the findings of my initial research along with a discussion on the most effective way of utilising image recognition in the project.

Sprint 2

The second sprint focused on the design and implementation of the core architecture of the bot. With cognitive architectures in mind, I outlined the main functions and interactions between components. This sprint also saw the training of the descriptor with all game objects within my test environment. Finally the descriptor was integrated into the previously implemented design, with the implementation able to recognise objects within scenes. The deliverable focused on the successful classification of objects within the SHERLOCK scenes.

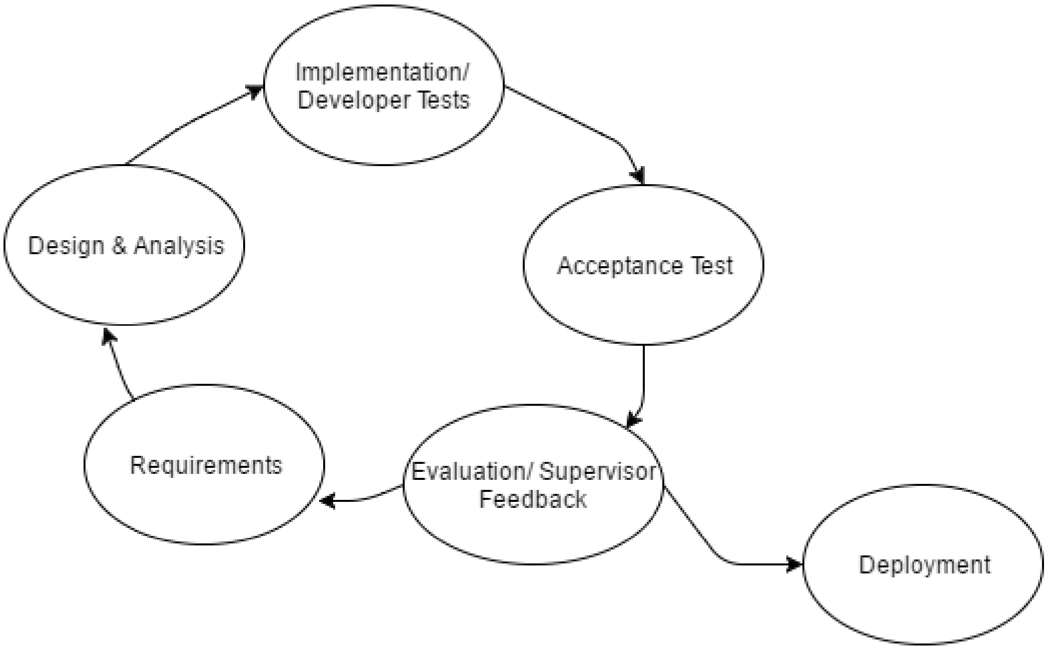
Sprint 3

The third sprint focused on the use of CENode and the SHERLOCK CE model to produce CNL output. This sprint saw the default CE model modified to integrate the training parameters and object images for the AI player to reason with. The deliverable in this sprint focused on the return of valid ITA CE string following a classification of an object within a scene.

Sprint 4

The fourth and final sprint focused on communicating the AI players CNL output with the central agent. This ensured it could satisfy the “one of the crowd” objective. This sprint saw collaboration with the CENode developer Will Webberley to ensure a successful integration. The final deliverable saw the finished implementation demonstrated successfully, with the SHERLOCK dashboard updated by the AI player.

Development Flow Diagram


Implementation/ Developer Tests
Acceptance Test
Evaluation/ Supervisor Feedback
Deployment
Requirements
Design & Analysis

The above diagram illustrates the development progress during sprints. The diagram attempts to visualise the iterative nature of development, with the requirements from the stakeholder being our starting point. Following the requirements comes the design and analysis of the sprint, where system flows and interactions were conceptualised. This was followed by the implementation of the design, with small scale developer tests to accompany. Acceptance tests followed, ensuring I'd met the deliverables set by my supervisor at the start of the sprint. The final step was to personally evaluate my work and receive feedback from my supervisor. This phase would see the deliverable critiqued with improvements, and changes suggested for for the next sprint. Once all supervisor specifications had been met, the implementation could be deployed.

3.2 Requirements

As with any software system a Software Requirement Specification is produced from stakeholder requirements. The SRS for this project includes several functional and nonfunctional requirements which concern the technical implementation. These requirements can also be used to evaluate the success of the implementation compared to the initial design.

3.21 Functional Requirements

1. **The system must be prompted to answer a question**

Acceptance Criteria

- A question in the form of valid textual input must be received and processed by the AI player.
- Only valid questions in the SHERLOCK game must be acted upon.

Justification

This functionality aims to mimic that of the human player being prompted to answer questions in the SHERLOCK game. It's important a valid question is provided or the AI player will be unable to gather the necessary resources to answer the question.

2. **The system must produce CNL output with relationships**

Acceptance Criteria

- A CNL output must be constructed in the form of valid ITA CE.
- The CNL output must have the relationship between both objects.

Justification

If the system does not produce a CNL output to answer the questions then it'll be unable to communicate what it sees to the shared knowledge base. This is due to CENode requiring interaction to be in the form of CE sentences wrapped in CE tell cards.

3. **The system must play the game alongside human players**

Acceptance Criteria

- The system must communicate its perceptions via tell cards to the central agent.
- An empty SHERLOCK dashboard listening to the central agent must change from grey to yellow when a valid tell card is posted.
- A human player must also send an observation to the central agent via the SHERLOCK conversational interface.
- The human player's observation must also change the colour of the SHERLOCK dashboard, with the ai players observation clearly visible.
- The human agent must “ask” its conversational agent a question regarding the AI players observation.

Justification

Having dashboard squares change colour from 2 different input sources, implies the central agent is relaying its knowledge to other players in the game. Operating as a human player, we should be able to see AI players observations change the status of the human players local SHERLOCK dashboard. Performing an “ask” to the conversational agent, will prove other players have access to the AI players observations.

4. **The system should use the same or similar model of the world as human players.**

Acceptance Criteria

- The AI player must have its own local user agent with the a version of the SHERLOCK CE model on it.
- This local user agent must provide the AI player with the same concepts, objects, rules and relationships available to human players.

Justification

Having a similar view of the world as a human player would allow the AI player to identify game objects and build relationships between objects.

5. **The system must not launch if a local CENode user agent is not operational**

Acceptance Criteria

- An error must appear informing that a local CENode agent is not operational, followed by the successfully exit of the system.

Justification

The system relies on a local CENode instance being operational, so queries regarding the environment can be made. The system can can only perform reasoning if it has access to the games’ concepts and instances hosted on this CENode instance. Without access, the system will cease to function correctly.

6. **AI player must use computer vision techniques to accurately perceive objects in a scene**

Acceptance Criteria

- We can deduce whether the computer vision technique has accurately perceived objects by observing the CNL produced relating to that scene.
- Each question's output should be manually observed to determine whether perception has been successful.

Justification

If the system is producing wrong observations then the crowdsourced knowledge base will contain inconclusive answers. By perceiving correctly, it would prove the game objects have been correctly trained, and that the shared knowledge is correct. While testing, each CNL string relating to a question will be manually observed for accuracy by a human tester.

3.22 Non Functional Requirements

1. **Invalid input must be handled gracefully**

Acceptance Criteria

- The system should catch invalid input and not proceed until valid input is given.
- Invalid input must return an informative error, specifying what type of input the system requires.

Justification

If the system is prompted with invalid input, having an error message informing what type of input the systems looking for, will save the user time in resolving the error.

2. **Unsuccessful HTTP requests must be handled gracefully**

Acceptance Criteria

- If a HTTP error is encountered, it must be caught and returned with an informative message on which endpoint is not responding. eg (If a localhost instance is not active or has last connection while the AI player is operational)

Justification

Having failed HTTP requests means relevant data isn't being sent or received into the system. This can adversely affect the core functionality of the system. Having an error message informing of the precise component a connection failure occurred in, can assist in fixing the issue.

3.23 Use Case

Use cases help us define the flow and consequence from a user interaction with the system. Due to the nature of this system, there is only a single use case from the perspective of a typical user. This use case will include a basic flow and alternative flows.

User prompts the system with a question

Basic Flow

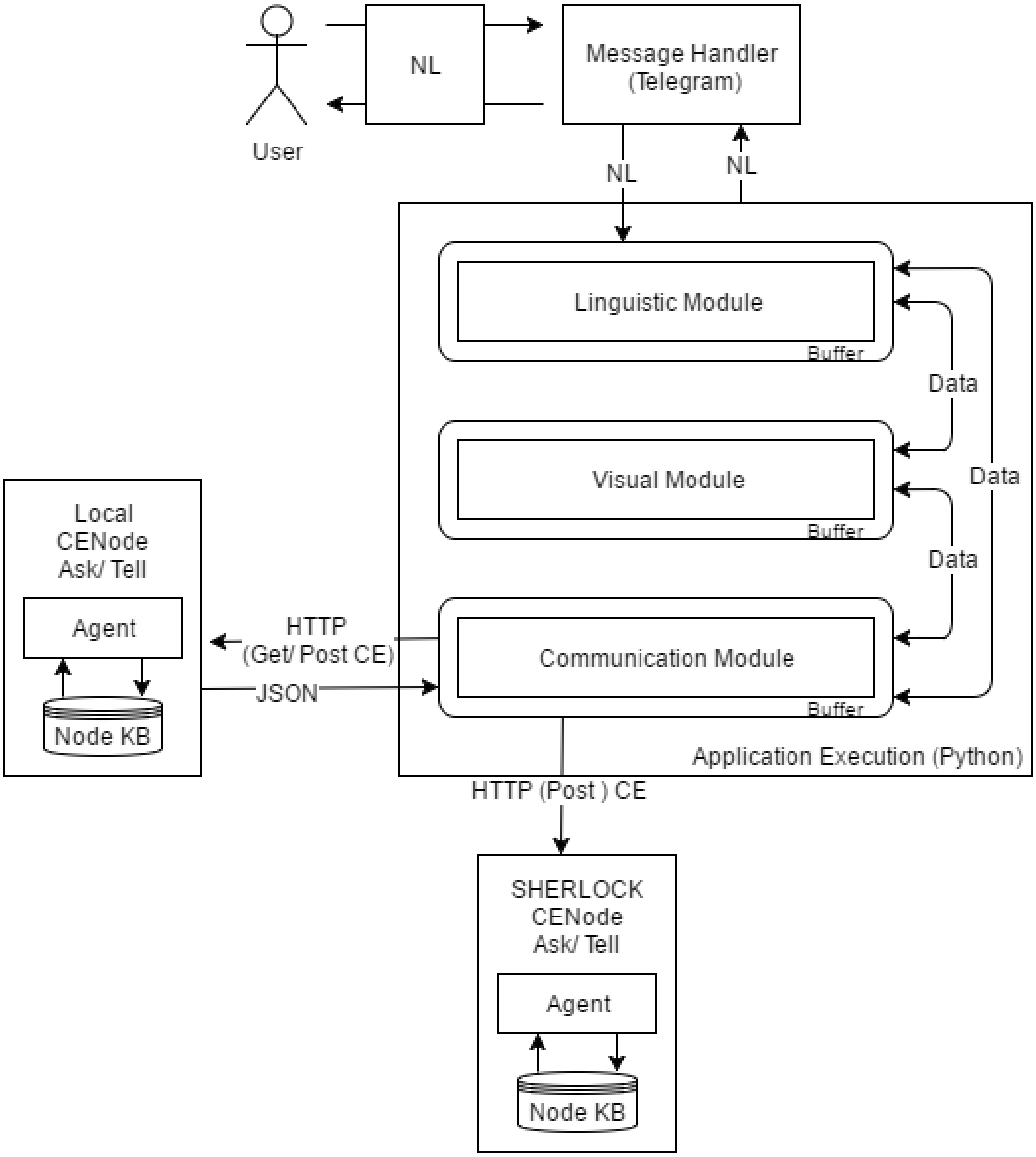
1. User prompts the system with a valid SHERLOCK question.
2. System acknowledges the question and reasons with it.
3. System scans it’s poster “feeds” with inputs derived from the question.
4. System returns its observation in CNL to the user.
5. System sends its observation to the shared KB.

Alternative flow

1. User prompts the system with invalid input.
2. System rejects the input and raises and error.
3. System returns to a “listening” state where it’s waiting for input.

3.3 System Architecture

The system architecture provides a conceptual model for defining the structure, behaviour and views of the system. With the aid of a high level diagram, I'll attempt to map the interactions between different components, along with the data exchanges that take place.



The above diagram illustrates a high level view of the systems architecture, while incorporating several characteristics found in the ACT-R CA. Each module has its own buffer which aims to store the state of the module at that time. In the context of the application, the buffers will be variable arguments passed between modules. Execution and production rules have been programmed into the application, whereas declarative memory can be accessed via a HTTP request to the local CENode agent.

The system's networking will be performed by the communication module. This module will perform HTTP Get and Post requests to both central and local agents. External data requests can be invoked by any function in the system by calling the communication module. The diagram shows 2 CEServer instances; the localhost CENode and the SHERLOCK CENode. The localhost CENode can will have the modified CE model posted to it when the AI player is launched. It can then be queried by the AI player via Get requests and receive a JSON response.The SHERLOCK CENode represents the central agent, where constructed CE output can be posted. The SHERLOCK CENode will be the central agent all players will post to. Centralising all CENode communication has made the module far more cohesive, and understandable from a developer's perspective. The exception to this cohesion is the telegram message handler, which handles its communication through an external API.

The systems visual reasoning will be performed by the visual module. This module will incorporate the SIFT algorithm previously discussed, and will take trained parameters and image paths as input. Given the correct inputs, the module will perform visual reasoning and return found objects.

The systems linguistic module will handle textual reasoning on NL inputs, and produce outputs in CE.

3.4 Modified SHERLOCK Model

When a SHERLOCK game is played, users agent will report back to a central CENode server instance. This instance will be running the SHERLOCK CE model, containing the SHERLOCK concepts, rules and entities of the game. For this project, I'll be using a modified version of the model with extra concepts and concept values, however the core functionality of the game will still be the same. In the original version of the model (Appendix 1), not every object had an image associated with it (eg: hat colour, sport). This is due to them not being labeled a “sherlock thing” which inherits the “imageable thing” concept. The below table illustrates the changes made to these non imageable concepts.

Old Model	Modified Model

`conceptualise a ~ hat colour ~ C`
`conceptualise a ~ sport ~ S`
`there is a hat colour named 'green'`
`there is a sport named 'tennis'`
	`conceptualise a ~ hat colour ~ C that is a sherlock thing`
	`conceptualise a ~ sport ~ S that is a sherlock thing`
	`there is a hat colour named 'green' that has 'green_hat.jpg' as image`
	`there is a sport named 'tennis' that has 'tennis.jpg' as image`

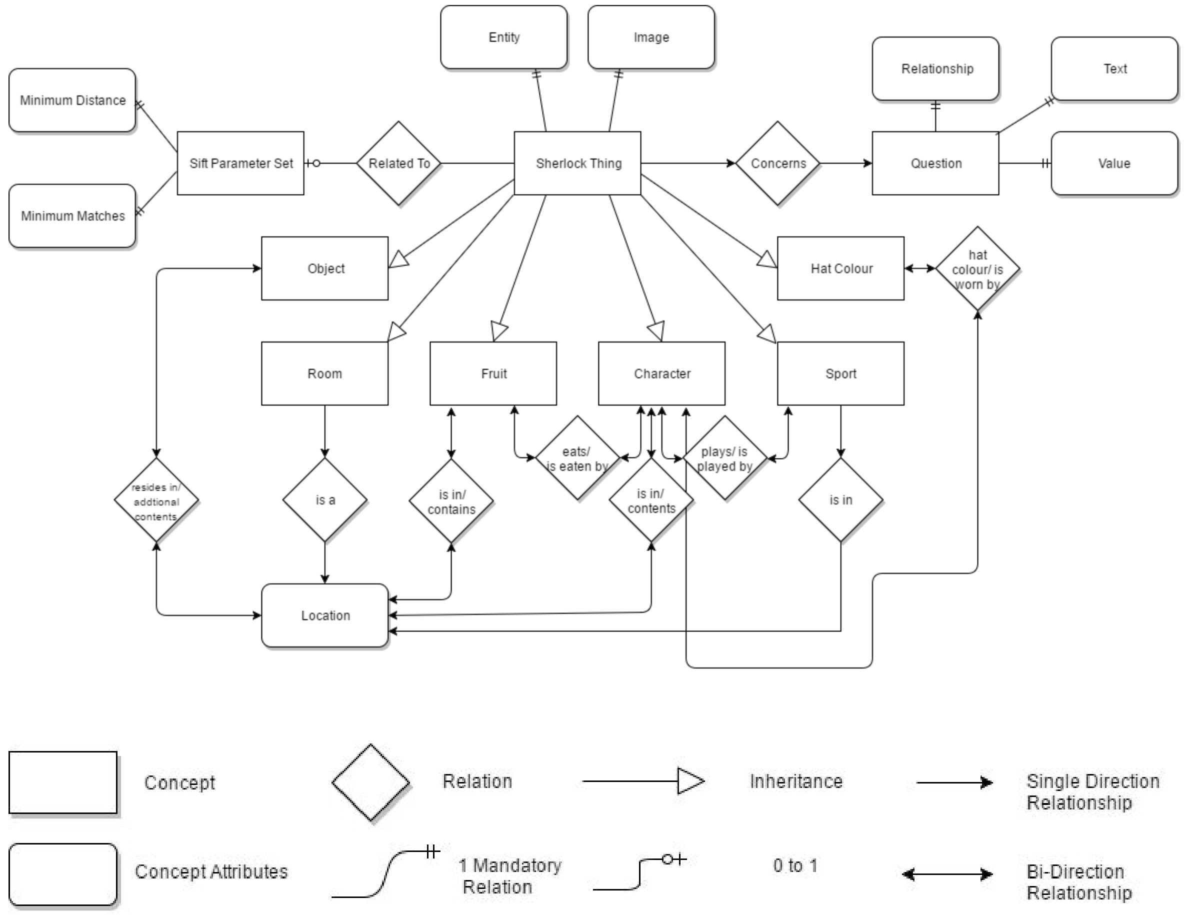
The old model also contained many legacy concepts from a previous version of the game. These concepts will need to be removed in the modified model.

My modified model (Appendix 2) clears out the legacy entities and makes every game object imageable. My model also introduces a new concept known as a “SIFT Parameter Set”. This parameter set is “related to” the “sherlock thing”, and shares a common name with objects. It is not however bound by an explicit relationship. Not binding the parameter sets explicitly to images, but still having a common name, assured the game's core functionality was still intact. The system could access the image parameter sets by referring to the object's name followed by the "parameters" keyword. (EG = "Elephant parameters"). The purpose of the parameter set concept was to store trained values relating to the SIFT classifier, in a single central location all data could be retrieved from.

SIFT concept

```
conceptualise a ~ sift parameter set ~ P that has the sherlock thing T as ~ related thing ~ and has
the value K as ~ maximum distance ~ and has the value J as ~ minimum matches ~

there is a sift parameter set named 'Elephant parameters' that has the character 'Elephant' as
related thing and has '45' as maximum distance and has '65' as minimum matches
```

Following is a conceptual visualising of my modified SHERLOCK model.



The relationship diagram above attempts to describe the model at a conceptual level, however the CE model differs greatly at the semantic level. The knowledge base in reality has a monotonic structure meaning cardinality is not possible. When a query is sent to the model, it'll return every attribute, relationship and value related to the object instance. The domain model allows the node to reason about its world and make inferences based on defined rules. It's this domain model that allows the knowledge base to be queried.

In total there are 9 core concepts defined within the model, which include: Sherlock thing, question, sift parameter set, object, room, fruit, character, sport and hat colour. These concepts represent different entities within the game, with some inheriting from others, some having relations with others, and some only representing their own attributes.

Transitive relations can be established between concepts through the use of rules. For example, a rule can instruct that a fruit "is eaten by" a character, however a character also "eats" a fruit. These transitive relationships between concept properties have the added benefit of being bi-directional. This means two different statements added to the KB can come to the same conclusion. For example "The character 'Zebra' eats the fruit 'apple'" and "The fruit 'apple' is eaten by the character 'Zebra'" are inverse statements that come to the same conclusion. Following the declaration of rules and concepts, instances of concepts can be specified. The model has an instance for every identifiable object within the game, as well as having a sift parameter set instance for each imageable object. The use of these instances will be further explained in the implementation.

4.0 Implementation

In this section I will give an in-depth overview of the systems implementation, including justifications for decisions made and challenges faced along the way. The code snippets provided aim to give an overview, however some aspects have been omitted. Each module class within the system will have an accompanying diagram describing the interactions between the class methods. The full code base can be found in Appendix 5.

The current implementation described below can play the game just like a human, in the context it can form impressions of the world and communicate them to a designated central agent. It however does not maintain its local agent, and instead interacts with the central agent by directly posting to it. The consequence of this is that the AI player cannot receive updates from other players in the game, however other players can receive the AI players observations to their local agent. The local agent in this implementation is used to gather resources from its CE model to construct CNL strings relating to the environment. In future versions, the AI player should able to maintain its local agent while receiving observations from other agents.

4.1 Development Environment

This section will see a quick overview into the environment the system was developed in, including hardware and software.

4.11 Hardware Environment

The hardware environment consisted of a quad-core i7-6700 running at 3.4GHz and 16GB of DIMM running at 2133 Mhz. The CPU supports 8 threads, with L1 cache of 256KB, L2 cache of 1.0MB and L3 cache of 8.0MB. All codes were executed in a Bash (version 4.3.11) environment running Ubuntu version 14.04. This specification proved sufficient to execute the SIFT algorithm in an acceptable time, while also hosting a local CENode server instance.

4.12 Software Environment

4.121 Primary Languages

The primary languages used in development were Python and Node.js. Python was used to implement the AI player itself, whereas node.js was the language CENode was implemented in. Bar some variable modifications, the CENode implementation used in this project was developed by Dr Will Webberley. Communication between both systems was through the HTTP protocol, specifically the Get and Post functions.

The 2 languages considered for the AI player implementation were Java and Python due to their bindings with the OpenCV image library. Although OpenCV is native to C++, my inexperience with the language along with support for Python and Java ruled it out for use in development. As these languages were simply wrappers to the original C++ implementation, there was no loss in performance when the Java or Python API's interacted with it.

Java and Python both have object oriented capabilities which is suitable for creating a modular implementation, however there are several key differences which influenced my decision to use Python over Java. Firstly Python is a portable language and is pre-installed on all current Linux distributions. If not for the external API's used by my implementation, the Python code could execute natively on all Linux machines. This is in comparison to Java which does not have the full language pre-installed on Linux machines, and can take extensive setup to get it working correctly. Secondly Python is supported by a centralised package management system known as pip [27], which provides a simple interface for installing external packages. Installing Java packages on the other hand can be quite difficult due to the absence of a centralised package manager. To install external resources, the user would have to download each JAR (Java Archive), and manually add it to the classpath configuration variable. As this project utilised several external libraries, having a streamlined and easy to use package manager would be beneficial. Thirdly, I'll be using OpenCV version 3.2, whose documentation is currently only up to date for the Python interface [29]. As OpenCV is an extensive package, it was crucial there was sufficient and up to date documentation available to reference when implementing it. Finally, from a personal perspective, my experience programming in Python greatly outweighs that of my development experience in Java; therefore I was far more confident I could produce quality software using Python rather than Java.

4.122 Further API's Used

Excluding OpenCV, the other Python packages deployed in this project were Requests[30] and Telepot [31].

Python has native HTTP functionality in the form of Urllib and Urllib2, however the HTTP library Requests was chosen to handle HTTP calls. There were several factors which influenced my decision to use Requests over the native urllib and urllib2 libraries. Firstly Requests fully supports the restful API which meant GET, POST, PUT and DELETE requests could be made to URLs. This was beneficial as the system will need to make GET and POST requests to CENode. Secondly, unlike urllib and urllib2, there's no need to manually encode data parameters. Urllib requires the '***urllib.encode()'*** function to encode data, whereas Requests simply requires a dictionary argument and encoding is automatically handled. An example of a POST request using the Python Requests API is as follows;

```
user_data = {"firstname": "Robert", "lastname": "Harris", "password": "Hello88"}

post_data = requests.post('http://www.example.com/user', data=user_data)
```

The code snippet above shows the dictionary defined in `user_data` set as the `data` variable in the POST request. the data in the `data` variable and will be automatically converted to unicode and posted to the address. As well as automatic encoding, it also contains automatic JSON decoding of GET requests. Finally the requests API contains a more elegant solution for error handling that urllib2. If authentication fails in urllib2 then a urllib2.Error is raised, however using Requests would see normal response object returned, omitting the need to implement an exception unless the developer explicitly wants to.

Telepot is an API used to build applications for the Telegram Bot API and is supported for Python 2.7 and 3.5. Initially the telepot API was incorporated as an image relay platform while prototyping with image classification techniques, however as the project progressed it proved a useful solution for interacting with the AI player. Using Telegram as a message relay, one could prompt the system to answer questions within the game and receive a response back. As it's also a cross platform messaging service, the system could be interacted with from a desktop or mobile device. Following are some example interactions with the bot during development;

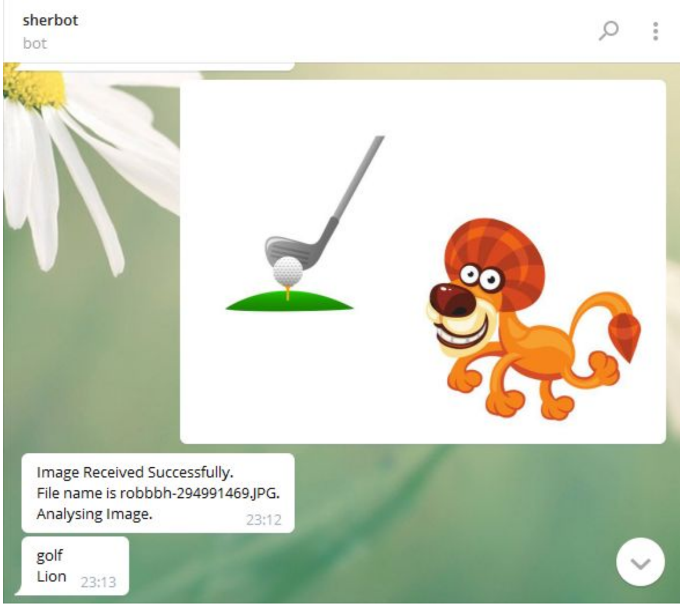
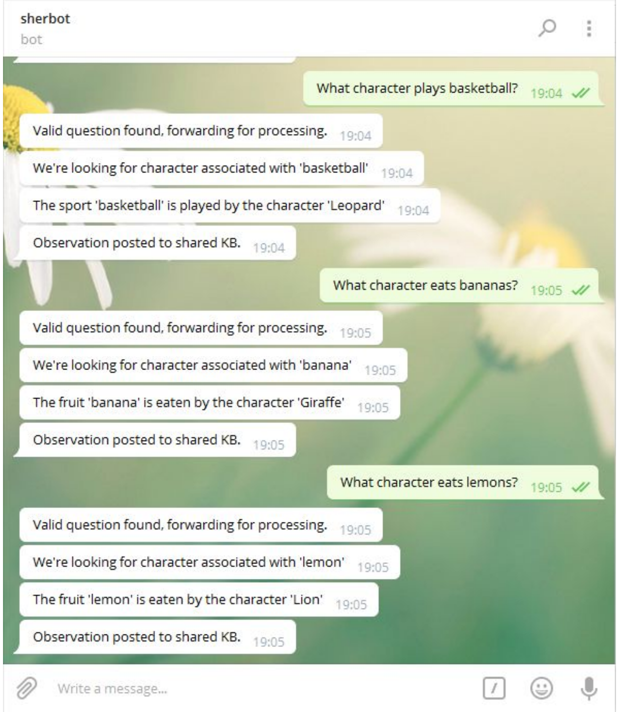


Image Received Successfully.
File name is robbbh-294991469.JPG.
Analysing Image. 23:12

golf
Lion 23:13

The above screenshot is from early in development when scene images could be posted to the bot and classifications could be made on the objects present. Below is a screenshot from a later build where a user can prompt the bot with a question.



What character plays basketball? 19:04

Valid question found, forwarding for processing. 19:04

We're looking for character associated with 'basketball' 19:04

The sport 'basketball' is played by the character 'Leopard' 19:04

Observation posted to shared KB. 19:04

What character eats bananas? 19:05

Valid question found, forwarding for processing. 19:05

We're looking for character associated with 'banana' 19:05

The fruit 'banana' is eaten by the character 'Giraffe' 19:05

Observation posted to shared KB. 19:05

What character eats lemons? 19:05

Valid question found, forwarding for processing. 19:05

We're looking for character associated with 'lemon' 19:05

The fruit 'lemon' is eaten by the character 'Lion' 19:05

Observation posted to shared KB. 19:05

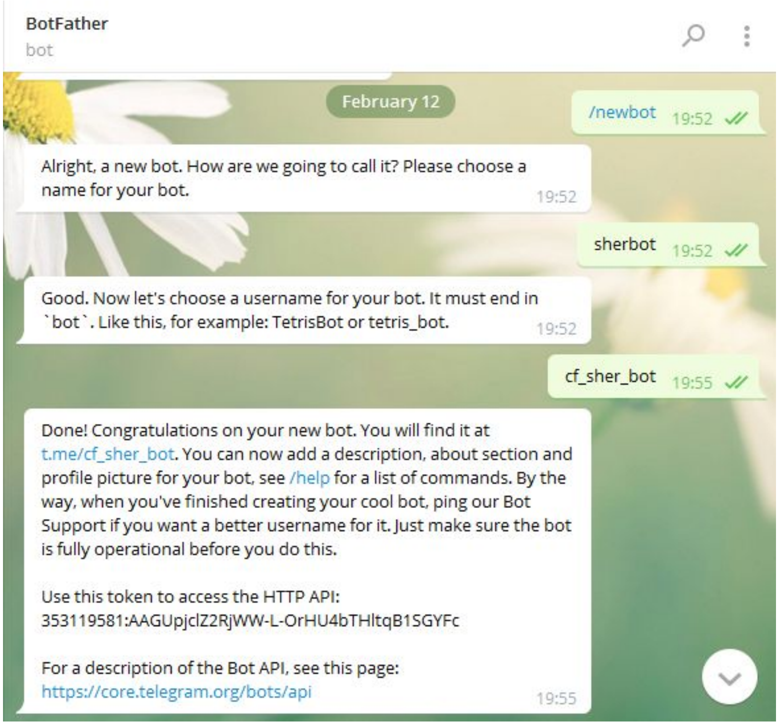
4.123 Software Versions

Software	Version
Python	2.7.6

OpenCV	3.2.0
Node.js	7.9.0
gcc	4.8.4
Requests	2.2.1
Telepot	9.1
Bash	4.3.11
CENode	3.0.7

4.2 Making and Implementing a Telegram Message Relay Bot

To create a bot using the Telegram API, one would have to receive a HTTP API token generated by Telegram's BotFather. Interaction with BotFather can be done through the Telegram App by sending the message `/newbot` to the username **@BotFather**. BotFather will then walk you through the initial steps to setting up your bot. The following screenshot shows the interaction with BotFather to setup the bot used in this project.



/newbot 19:52

Alright, a new bot. How are we going to call it? Please choose a name for your bot. 19:52

sherbot 19:52

Good. Now let's choose a username for your bot. It must end in `bot`. Like this, for example: TetrisBot or tetris_bot. 19:52

cf_sher_bot 19:55

Done! Congratulations on your new bot. You will find it at t.me/cf_sher_bot. You can now add a description, about section and profile picture for your bot, see /help for a list of commands. By the way, when you've finished creating your cool bot, ping our Bot Support if you want a better username for it. Just make sure the bot is fully operational before you do this.

Use this token to access the HTTP API:
353119581:AAGUpjclZ2RjWW-L-OrHU4bTHltqB1SGYFc

For a description of the Bot API, see this page:
https://core.telegram.org/bots/api 19:55

As we can see from the screenshot, our generated HTTP API key is `353119581:AAGUpjclZ2RjWW-L-OrHU4bTHltqB1SGYFc`. Now our bot is set up, we can implement the telepot API in Python to enable interaction. The following code snippet shows how the `telegram_handle` class sets up the bot interaction.

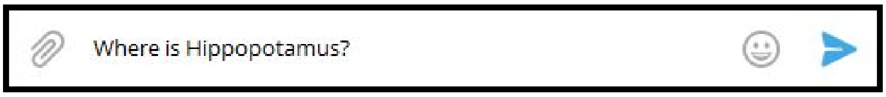
```
class telegram_handle:

    def __init__(self):
        # bot API key
        self.bot = telepot.Bot('353119581:AAGUpjclZ2RjWW-L-OrHU4bTHltqB1SGYFc')
        self.bot.message_loop(self.handle)
        self.trusted = ["robbbh"]
```

In the constructor we specify the HTTP key in the `telepot.Bot` function, along with a message looper and trusted users.

The `bot.message_loop(self.handle)` takes a method as input (in this case *handle*) and spawns a new thread. This thread is responsible for handling that particular message, and will provide the *handle* method with the `msg` argument containing a JSON message packet. The telepot documentation gives the following explanation on its functionality;

"Spawn a thread to constantly getUpdates or pull updates from a queue. Apply callback to every message received. Also starts the scheduler thread for internal events."

If a message is sent to the bot then a message packet will be received by the system. This packet can be accessed in the `msg` variable and will be encoded in a JSON format. Sending the bot the following message,


Where is Hippopotamus?

will produce the following packet;

```
{u'date': 1492951430, u'text': u'Where is Hippopotamus?', u'from': {u'username': u'robbbh',
u'first_name': u'Robert', u'last_name': u'Harris', u'id': 294991469}, u'message_id': 2411,
u'chat': {u'username': u'robbbh', u'first_name': u'Robert', u'last_name': u'Harris',
u'type': u'private', u'id': 294991469}}
```

The following screenshot shows the handle method;

```
   def handle(self, msg):
        self.userName = msg["from"]["username"]
        self.chat_id = msg['chat']['id']

        if self.userName not in self.trusted:
            self.bot.sendMessage(self.chat_id, "Unverified User...\nPlease use a
verified account.")
            return
        else:
            self.check_message_type(msg)
```

The handle method will retrieve the packets username and chat_id, and check if the username is in the approved list. For this project, my Telegram username (robbbh) will be the only name in the approved list. The chat_id variable will contain the conversations identifier, enabling the bot to send a reply back to the user. A response can be sent back to the user, using this function and its arguments;

```
self.bot.sendMessage(self, chat_id, “FooBar”)
```

In the handle method this is used to inform a rejected username that they are not in the approved list of users. This chat_id is unique between the user and the bot, and will need to be accessible by all objects to send messages back to the user.

Once the username has been approved, it’s passed to the `check_message_type` function which will check the bot has received the correct message type. This is necessary as it’s possible to send many types of media to the bot, however we’re only looking for messages of the type text. The packet contains information on the message type, so we can simply check for the keyword `text` in the packet. If the message is of textual type, then it’s passed into the linguistics module for processing. If not then an error is raised, and the bot will return to a “listening for input” state. The code snippet below represents this functionality.

```
def check_message_type(self, msg):

     if "text" in msg:
         self.message = msg['text']
         print "Recieved text input = '{0}'".format(self.message)

         interpret_text = Linguistic_Module(self.chat_id)
         interpret_text.determine_question_type(str(self.message))

     else:
         error_message = "No valid message type found."
         print error_message
         self.bot.sendMessage(self.chat_id, error_message)
         return
```

4.3 Setting up CENode Locally

In this section I’ll give an insight into how a local version CENode was installed and how the system interacted with it as a means of environmental understanding .

The first step was to clone the CENode project onto my system using the following command `git clone git@github.com:flyingsparx/CENode.git`. With the CENode repository successfully cloned, the next step was to launch a local CENode server instance that could be interacted with. This could be achieved by executing the CEServer file inside the CENode directory with the following terminal command; `node src/CEServer.js Moira 5000 core`. This command would launch a server instance with an agent named Moira, listening on port 5000, with the core CE model launched. The core model in this context contains basic model concepts such as cards, policies and rules. With the server instance operational on port 5000, I could now query it from the terminal with the `curl` command.

Executing `curl http://localhost:5000/concepts` will return all concepts in the core model in JSON-encoded format;

```
[{"name":"entity","id":1},{"name":"imageable thing","id":2},
{"name":"timestamp","id":3}, {"name":"agent","id":4}, {"name":"individual","id":5},
{"name":"card","id":6}, {"name":"tell card","id":7}, {"name":"ask card","id":8},
{"name":"gist card","id":9}, {"name":"nl card","id":10}, {"name":"confirm
card","id":11}, {"name":"location","id":12}, {"name":"locatable thing","id":13},
{"name":"rule","id":14}, {"name":"policy","id":15}, {"name":"tell policy","id":16},
{"name":"ask policy","id":17}, {"name":"listen policy","id":18}, {"name":"listen
onbehalfof policy","id":19}, {"name":"forwardall policy","id":20}, {"name":"feedback
policy","id":21}]
```

Executing `curl http://localhost:5000/instances` will return all instances in the core model in a JSON-encoded format;

```
[{"name":"Moira","id":1,"conceptName":"agent","conceptId":4}]
```

Now we have verified the server is launched and can be successfully contacted, we can upload our SHERLOCK model to it. Executing the following command will POST the model to the local server.

```
curl --form "fileupload=@sherlock.ce http://localhost:5000/sentences
```

Re-executing the query for concepts and instances should return a far more extensive JSON output.

`curl http://localhost:5000/concepts` will now return;

```
[{"name":"entity","id":1}, {"name":"imageable thing","id":2},
{"name":"timestamp","id":3}, {"name":"agent","id":4}, {"name":"individual","id":5},
{"name":"card","id":6}, {"name":"tell card","id":7}, {"name":"ask card","id":8},
{"name":"gist card","id":9}, {"name":"nl card","id":10}, {"name":"confirm
card","id":11}, {"name":"location","id":12}, {"name":"locatable thing","id":13},
{"name":"rule","id":14}, {"name":"policy","id":15}, {"name":"tell policy","id":16},
{"name":"ask policy","id":17}, {"name":"listen policy","id":18}, {"name":"listen
onbehalfof policy","id":19}, {"name":"forwardall policy","id":20}, {"name":"feedback
policy","id":21}, {"name":"sherlock thing","id":22}, {"name":"fruit","id":23},
{"name":"room","id":24}, {"name":"hat colour","id":25}, {"name":"sport","id":26},
{"name":"character","id":27}, {"name":"object","id":28}, {"name":"question","id":29},
{"name":"sift parameter set","id":30}]
```

whereas `curl http://localhost:5000/instances` will return an extensive output which can be seen in Appendix 3.

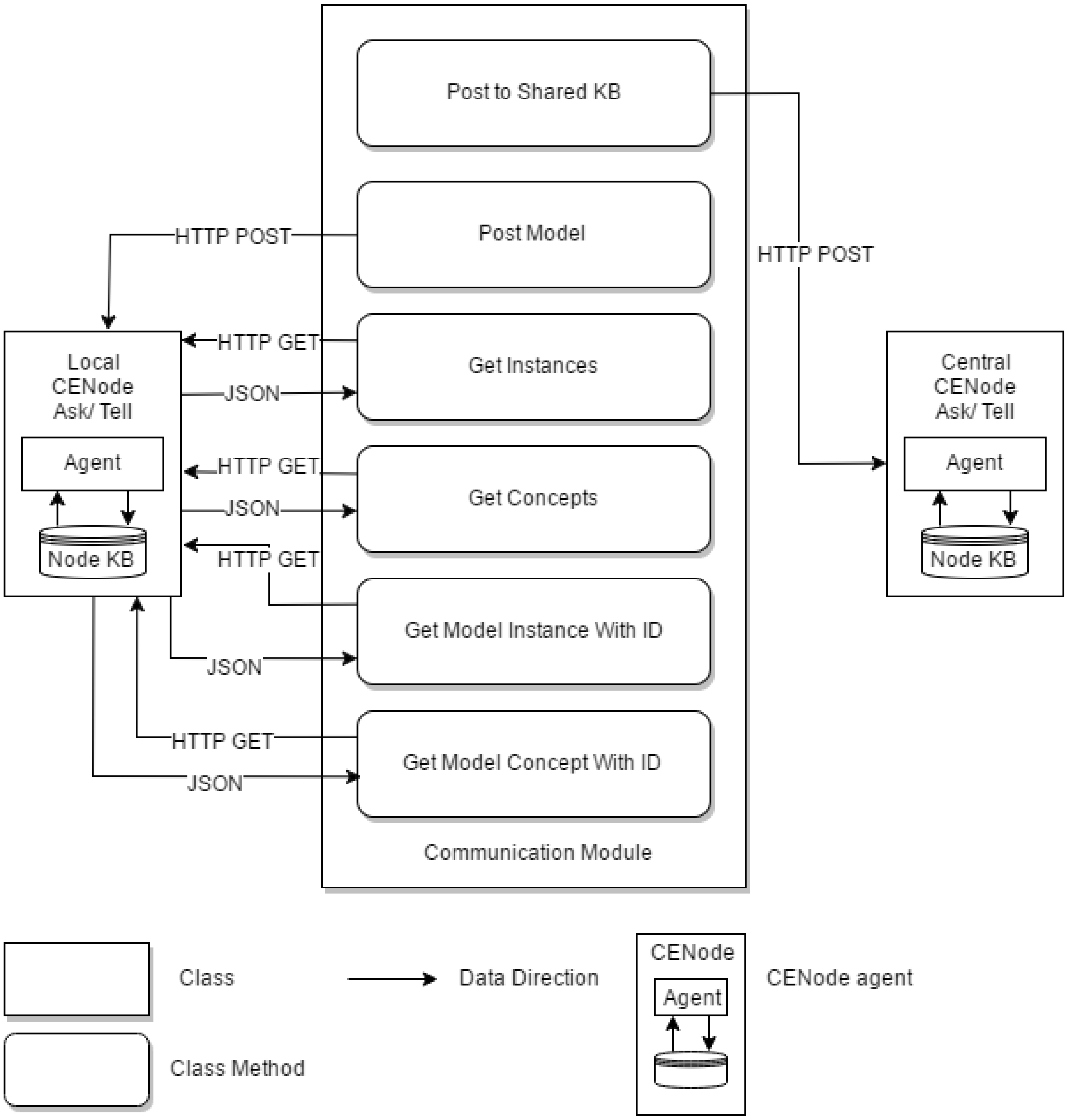
It's this process of querying instances and concepts, that the system can gather the correct resources it needs to reason with game objects and its environment. For example, querying the Giraffe character by its instance ID of 13 can be achieved with the following command `curl http://localhost:5000/instance?id=13`. It will then return the following JSON-encoded response.

```
{"name":"Giraffe", "conceptName":"character", "conceptId":27, "ce":"there is a character
named 'Giraffe' that has 'giraffe.jpg' as image.", "synonyms":[], "subConcepts":[],
"values":[{"label":"image", "targetName":"giraffe.jpg"}], "relationships":[]}
```

By querying the Giraffe instance we now have such information as concept type, concept id and image path. The system can reason with this type of information and attempt to build inputs for the image classifier and resources for building CNL strings.

4.4 The Communication Module

In the previous section I demonstrated how CENode could be interacted with through shell HTTP commands to gather game resources. In this section I'll be applying the same concept, but within the Python environment, and using the Requests API to execute HTTP commands. The communication module is where all interaction with CENode is handled by the system. The following diagram attempts to give a high level view of the communication modules interactions with external data resources.



In this implementation, the communication module is responsible for posting models, posting to the central agent and performing various get requests for model data. It consists of a single POST call to the central agent (by-passing the players "user agent"), along with a POST and 4 GET requests to the local agent. Each request is encapsulated with the `requests.exceptions.RequestException` to provide a concise error output for connection failures.

The module contains a simple constructor containing the port number for the localhost instance.

```
class Communincation_Module:

    def __init__(self):
        """ Performs system communication through the Requests API"""
        self.port = 8004
```

The first POST call is performed when the system is launched, and takes the form of the `post_model` method.

```
if __name__ == "__main__":
    Communincation_Module().post_model()
```

This method attempts to post the CE SHERLOCK model to the local CENode server instance. If the POST is unsuccessful then the system will not launch and an error will be thrown. User prompts and game data reasoning wouldn't be possible without a the local CENode hosting the game model, therefore the system will shutdown if one is not operational.

```
   def post_model(self):
        try:
            with open('sherlock.ce', 'r') as sherlock_model:
                model = sherlock_model.read()
            requests.post('http://localhost:' + str(self.port) + '/sentences', data =
model)
        except requests.exceptions.RequestException as e:
            error_message = "Error posting model to instance. Please launch the
localhost CEServer to continue."
            print error_message
            print e
            sys.exit(1)
```

The above code snippet illustrates how this functionality was implemented using the Requests API. We can see the method reads the model from the `sherlock.ce` file, and proceeds to post it to the local CENode server instance.

The second POST call regards the posting of CE tell cards to the central CENode agent. It's this agent all players report their observations too. The current central agent is hosted on http://explorer.cenode.io with the agent name 'sherlock', operating on port 6789. The below code represents this posting functionality to the shared KB using the requests API.

```
   def post_to_shared_kb(self, tellcard):
        try:
            requests.post("http://explorer.cenode.io:" + "6789" + "/sentences", data =
tellcard)
        except requests.exceptions.RequestException as e:
            print "Posting to shared knowledge base failed. Please check the instance
exists at http://explorer.cenode.io."
            print e
```

The communication module contains 4 GET requests which the system regularly calls to retrieve data. The below code demonstrates the 4 GET calls using the Requests API. The first two return all instances and concepts within the model, whereas the latter two return unique instances and concepts specified by an ID.

```
   def get_instances(self):
        try:
            response = requests.get('http://localhost:' + str(self.port) +
'/instances').json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get instances. Please check the localhost CENode is
operational"
            print e
            os_.exit(0)


    def get_concepts(self):
        try:
            response = requests.get('http://localhost:' + str(self.port) +
'/concepts').json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get concepts. Please check the localhost CENode is
operational"
            print e
            os_.exit(0)


    def get_model_concept_with_id(self, id):
        try:
            response = requests.get("http://localhost:" + str(self.port) + "/concept" +
"?id=" + str(id)).json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get concept ID. Please check the localhost CENode is
operational and the concept ID exists."
            print e
            os_.exit(0)


    def get_model_instance_with_id(self, id):
        try:
            response = requests.get("http://localhost:" + str(self.port) + "/instance" +
"?id=" + str(id)).json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get instance ID. Please check the localhost CENode is
operational and the instance ID exists."
            print e
            os_.exit(0)
```

These calls would all return JSON-encoded responses as demonstrated in the previous section.

I was pleased with the implementation of the communication module as it abided by the single responsibility principle. This object-orientated principle states that a module or class should have responsibility over a single part of functionality. By following this principle, I've made the class more robust, maintainable and recyclable to other classes calling it.

4.5 The Visual Module

In this section I'll discuss implementation of the visual module, including the installation of the OpenCV package, the training of the descriptor and the implementation of the matcher. The visual module incorporates a forward chaining approach to perceiving the environment, rather than backward chaining. The system is task focused meaning it'll only remember an object if it's asked to look for it. Although a backward chaining approach would work in principle, the forward chaining approach was chosen due to its dynamic nature. For example, in a real world setting, the visual module may be connected to a live feed of an environment. As a real world environment can be changeable, a dynamic approach could catch these changeable events, whereas a backward chaining approach may miss them.

4.51 Installing OpenCV

Installing OpenCV was quite a difficult task due to the numerous dependencies the package relies on. Following a guide found at *pyimagesearch.com* [32] I did manage successfully install version 3.2.0 of the OpenCV package. Originally I had installed OpenCV version 3.1.0, however the Python bindings contained a bug when calling the Flann matcher function used in my implementation. The following error was thrown;

```
OpenCV Error: Assertion failed (The data should normally be NULL!) in allocate, file
opencv/modules/python/src2/cv2.cpp, line 163
Traceback (most recent call last):
  File "test.py", line 21, in <module>
    matches = flann.knnMatch(des1, des2,  k=2)
cv2.error: opencv/modules/python/src2/cv2.cpp:163: error: (-215) The data should
normally be NULL! in function allocate
```

Following research online, the bug was traced to the Python wrapper calling the `FlannBasedMatcher::add` overload in the C++ source code [33]. With the bug patched in OpenCV 3.2.0, updating my package to 3.2.0 addressed the issue.

4.52 Training the Images

One of the key elements to achieving an accurate classification for game objects, was training the classifier. My implementation used a maximum distance parameter and a minimum matches parameter as thresholds.

The maximum distance parameter refers to the maximum squared euclidean distance between the query image and training image. The lower the distance value, the more likely it is that the 2 images have similar properties. The matcher used in this implementation is the FLANN (Fast Library for Approximate Nearest Neighbors) matcher [35] which provides a collection of optimised algorithms for fast nearest neighbor search in large datasets. The Brute Force matcher was considered for this implementation, however the FLANN matcher was chosen due to its superior speed. The FLANN matcher yields its speed from an efficient k-dimensional tree data structure, and its use of an approximate nearest neighbour form of matching. Although it's not as accurate as the brute force approach, it has proved sufficient for this implementation.

Having a maximum distance threshold, allowed me to filter out dissimilar descriptors. The overall distance value is calculated by taking the sum of the top 20 best distances (shortest distances) and dividing them by 20. Following some experimentation, the top 20 distances were chosen, as it contained a good sample of best distances while filtering out false positives. Although some classifications contained less than 20 matches, their average distance was still too high to be considered a positive match.

The minimum matches parameter refers to the number of matches the Flann matcher produces. Generally a positive match will have a higher number of matches, however is not a conclusive measurement by itself. Coupling this with the distance parameter has allowed me to successfully identify game objects within scenes.

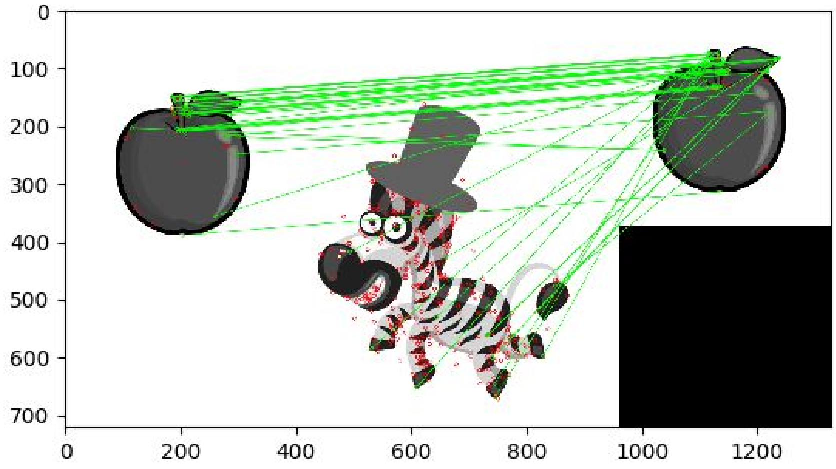
Following are the distances and matches produced by the *apple* object when tested against all poster scenes.

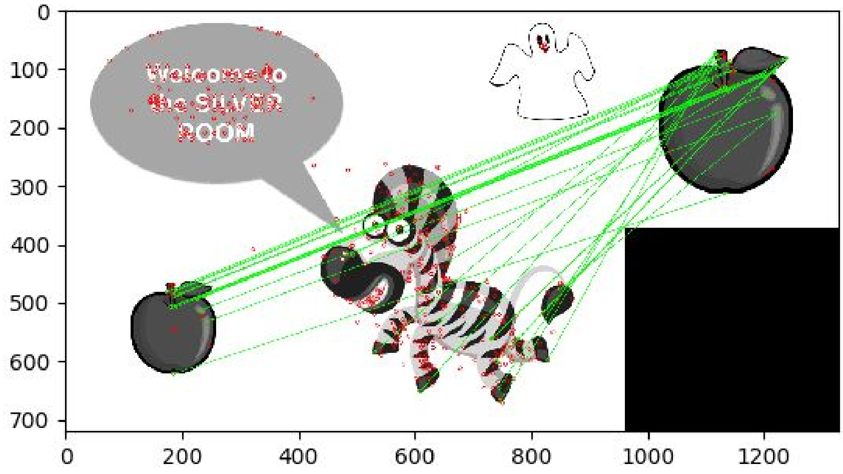
Query Image Name	**Training Image**	**Number of Matches**	**Distance**
elephant_orange.JPG	apple.jpg	8	92.903792572
elephant_orange_room.JPG	apple.jpg	10	135.497565079
elephant_tennis_greenhat.JPG	apple.jpg	11	123.690659904
giraffe_banana_redhat.JPG	apple.jpg	15	154.159870148
giraffe_room_redhat_robot.JPG	apple.jpg	21	220.922873306
giraffe_rugby.JPG	apple.jpg	17	171.832937622

hippo_pineapple.JPG	apple.jpg	16	198.512023163
hippo_room_bluehat.JPG	apple.jpg	20	267.396146393
hippo_soccer.JPG	apple.jpg	9	119.771272278
lion_golf.JPG	apple.jpg	23	186.96775341
lion_pinkhat.JPG	apple.jpg	34	133.380025291
lion_room_lemon_ape.JPG	apple.jpg	50	151.138672829
tiger_basketball_yellowhat.JPG	apple.jpg	16	172.676062393
tiger_pear_green_baloon.JPG	apple.jpg	37	122.598890114
tiger_pear.JPG	apple.jpg	38	116.710225105
zebra_apple_purplehat.JPG	apple.jpg	56	19.2249680042
zebra_apple_room_ghost.JPG	apple.jpg	33	67.7635392189
zebra_cricket.JPG	apple.jpg	22	213.569628906

The apple object is present in the *zebra_apple_purplehat.JPG* and *zebra_apple_room_ghost.JPG* scenes. Observing the table we can see both scenes have a much lower distance value of 19.2 and 67.8, followed by a match count of 56 and 33 respectively. We can see that *zebra_apple_purplehat.JPG* has a higher match count and a lower distance measurement than the *zebra_apple_room_ghost.JPG* scene even though they contain the same apple object. This is due to the apple object present in the *zebra_apple_room_ghost.JPG* scene having a smaller resolution than the apple training image. There is enough disparity however, between the other scenes that the apple object can be classified successfully. A successful classification could be achieved with a maximum distance of 90 and a minimum match count of 30. This process was repeated for every object in the game, until all objects had been trained for the descriptor.

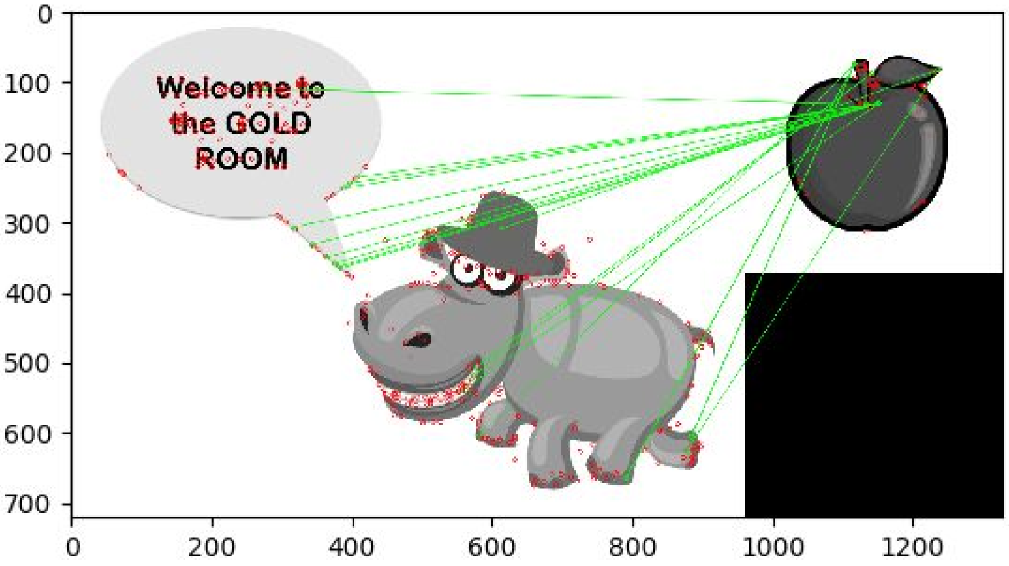
The following images visualise the *zebra_apple_purplehat.JPG* and *zebra_apple_room_ghost.JPG* keypoints being matched to the apple object. The images are processed in greyscale as the detector relies on the luminance of the image for detecting visual features. A greyscale image is just as good for detecting intensity as an RGB image, however its processing costs are greatly reduced due to it only comparing simple scalar algebraic operators (+ , -).





We can see that keypoints are being matched between the apple training image on the right and the apple object within the scene to the left, however there are several false positives present on the Zebra character in both scenes. Taking the top 20 best distances as discussed earlier, we can attempt to filter out these false positives.

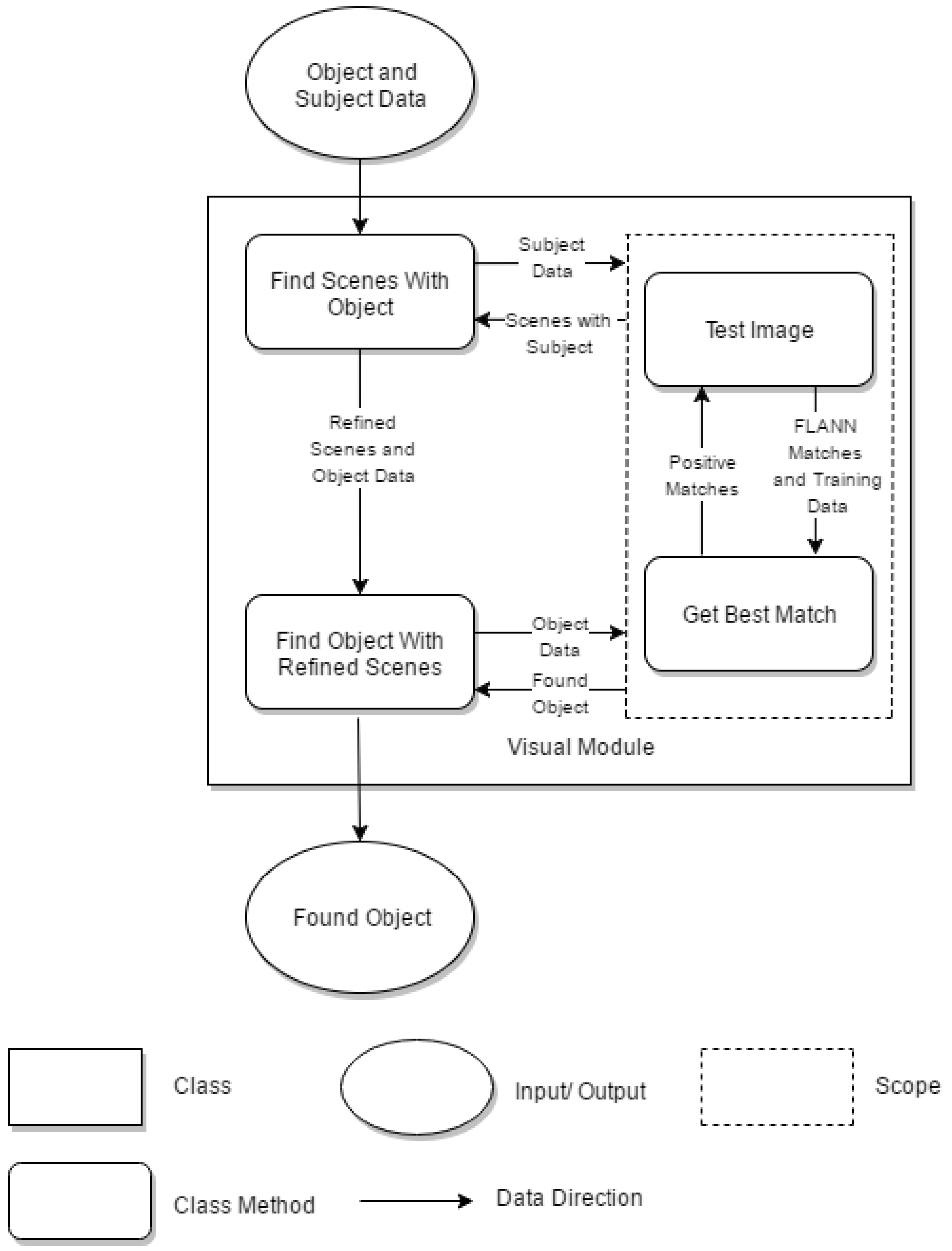
The table contains classifications for all scenes, with the majority not containing the apple object. The SIFT descriptor still however, attempts to match with keypoints present in both scenes. We can see in the image below how the SIFT descriptor has matched keypoints found in the apple image to keypoints found in the *hippo_room_bluehat.JPG*. The matcher found a total of 20 matches between the 2 images, with a distance average of 267.4. Using the trained parameters discussed above, the system can conclude that the apple object is not present in the *hippo_room_bluehat.JPG* scene. Training parameters for all objects can be found in the modified CE model in Appendix 2.



4.53 The Visual Module Class

The visual module class contains 4 methods overall, with each method interacting with each other to produce a classification. In this implementation, the system can access its “feed” to the posters through the ./*scenes* directory in the codes repository. In a real life environment we could assume this “feed” will be a connection to a camera system with access to the environment. (eg: A CCTV camera in an inaccessible location).

The interaction between methods can be visualised by the following diagram;



The module takes 2 inputs in the form of Object and Subject Data. These inputs will be in a tuple data structure containing an object image path, object image name, maximum average flann distance and minimum match value. The subject data will contain a single tuple with information about a single entity within the game (eg: The Elephant Character). The object data will contain a list of tuples containing information about a specific object within the game (eg: Fruit object = Apple, Pineapple, Bananas, Orange, Lemon, Pear).

The system will first interact with the *find_scenes_with_objects* method, which searches the environment in an attempt to find scenes containing the subject image. To do this, it'll extract the data contained in the *subject_data* tuple, and pass it to the *test_image* method. The *test_image* method will classify the subject image and scene images with the SIFT detector, before passing it on to the `get_best_match` method. The `get_best_match` method will return all scenes containing the subject image. A total of 18 iterations to the SIFT classifier will be completed in this first step, as there are 18 individual scenes in the game.

Now all positive scenes have been found, they're passed to the `find_objects_with_refined_scenes` method along with the object data. As mentioned previously, `object_data` contains a list of tuples where each tuple will have be iterated over and have its data extracted. Each iteration will extract the tuple data and pass it to the `test_image` method. As we have refined the scenes, the SIFT descriptor will have to iterate over a maximum of 3 scenes searching for a single object. Once the `get_best_match` has found a positive object, the object can be returned to the external module which requested classification.

Generally there will be 6 different objects the system will search for within the refined set of scenes. Assuming there are 3 scenes in the set of refined scenes, this process will complete a maximum of 18 iterations over the SIFT descriptor. Overall the SIFT descriptor will be utilised a maximum of 36 times when answering a question, however this number is relative to the number of refined scenes. Questions can produce up to 3 refined scenes, meaning the SIFT descriptor can be utilised 24, 30 or 36 times depending on the question.

4.54 Example Case

Given the question '*Where is Hippopotamus?*' the visual module would receive the following *object_data* and *subject_data* variables.

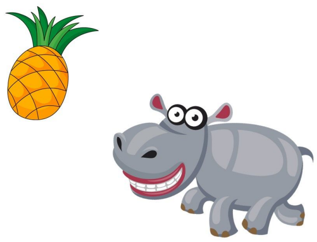
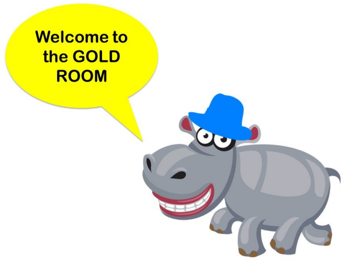
`subject_data = ('hippo.jpg', u'35', u'105', 'Hippopotamus')`

`object_data = [('ruby_room.jpg', u'10', u'155', 'Ruby Room'), ('saphire_room.jpg', u'5', u'200', 'Sapphire Room'), ('gold_room.jpg', u'30', u'155', 'Gold Room'), ('amber_room.jpg', u'32', u'175', 'Amber Room'), ('emerald_room.jpg', u'35', u'190', 'Emerald Room'), ('silver_room.jpg', u'5', u'190', 'Silver Room')]`

Observing the `subject_data` output, we can see it contains a tuple regarding the 'Hippopotamus' character. As the question has prompted for information about the Hippopotamus character, the system has determined that it is our "subject". The `object_data` variable on the other hand contains a list of objects that are concerned with the subject . As the question is prompting for a location, all of the game's locations have been returned. We can see the tuples returned in both variables have data concerning the object image, minimum matches, maximum distance and object name.

Now our variables have been determined, they can be sent for classification. The first step sees the system search for the Hippopotamus character within the environment. This would involve iterating through every poster scene, and returning each scene containing the subject. This example returned the following scenes;

```
scenes_with_object = ['scenes/hippo_pineapple.JPG',
'scenes/hippo_room_bluehat.JPG', 'scenes/hippo_soccer.JPG']
```

hippo_pineapple.JPG	hippo_room_bluehat.JPG	hippo_soccer.JPG
		

The next step sees the above scenes classified against the locations in *`object_data`*. This would ultimately produce the final classification, which would answer the initial question. The answer of ‘Gold Room’ is produced in this example.

4.55 Descriptor and Matcher Code

In this section I’ll discuss the implementation of the SIFT descriptor and FLANN matcher, along with the process of finding the best matches. The initial implementation took influence from the feature detection tutorials found in the OpenCV documentation [34].

Following is the `test_image` method which applies the SIFT algorithm and FLANN matcher.

```
def test_image(self, test, train, min_match, min_dist, input_image_name):
     """ Performs image classification with the SIFT descriptor """
     # opens query and train image in grayscale
     query_image = cv2.imread(test,0)
     train_image = cv2.imread(train,0)

     # Initiate SIFT detector
     sift = cv2.xfeatures2d.SIFT_create()

     # find the keypoints and descriptors with SIFT
     kp1, des1 = sift.detectAndCompute(query_image,None)
     kp2, des2 = sift.detectAndCompute(train_image,None)

     FLANN_INDEX_KDTREE = 0
     index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
     search_params = dict(checks = 50)

     flann = cv2.FlannBasedMatcher(index_params, search_params)

     matches = flann.knnMatch(des1,des2,k=2)

     return self.get_best_match(matches, test, train, min_dist, min_match,
 input_image_name)
```

The initial step is is to read in the query and training image into the system, and convert them to greyscale. This is done by using the `cv2.imread()` function while supplying the image path and integer flag. We use the flag value of 0 which specifies greyscale conversion. The next step is to initiate the SIFT detector with the `cv2.xfeatures2d.SIFT_create()` function. Initiating this function allows us to access its `detectAndCompute()` methods which will find keypoints and descriptors within the images in a single step.

The next step is to perform descriptor matching with the FLANN based matcher. The FLANN based matcher requires two dictionaries; one to specify which algorithm to be used, and one to specify related parameters. The `index_params` variable specifies that the `FLANN_INDEX_KDTREE` algorithm is to be used along with a set of 5 parallel kd-trees. The `search_params` variable specifies the number of times the trees in the index should be recursively traversed. The higher the value, the more precise the matching will be, however it'll increase the run time. 50 was deemed an acceptable number of checks as it was precise, and relatively quick. Now our parameters has been specified, the FLANN based matcher along with its arguments can be called with `cv2.FlannBasedMatcher(index_params, search_params)`. We could perform the matches with the `knnSearch` function available in the Flann Based matcher object. Using `matches=flann.knnMatch(des1,des2,k=2)` we can perform a K-nearest neighbour search for given query points. The `k=2` specified is the count of best matches found per query descriptor, and allows us to perform Lowe's ratio test which will be discussed below. The matches between both descriptors are saved to the `matches` variable ready to be filtered and reasoned with for positive and negative matches. This would be performed in the `get_best_match()` method as specified below.

```
    def get_best_match(self, matches, test, train, min_dist, min_match,
 input_image_name):
        """ Works out the best match based on training data and ratio test. """

        # store all the good matches as per Lowe's ratio test.
        good = []
        for m,n in matches:
            if m.distance < 0.7*n.distance:
                good.append(m)

        if len(good) >= int(min_match):
            match_array = [matches.distance for matches in good]
            # sorts matches by size
            matches = sorted(match_array, key = lambda x:x)
            # gets average of 20 best matches
            distance = sum(matches[:20])/20

            if distance <= int(min_dist):
                scene_path = test
                image = input_image_name
                return scene_path, input_image_name
```

There are 3 main steps the `get_best_match()` method executes to come to a conclusion. The first step is to perform Lowe's ratio test which is specified in David G. Lowe's paper, Distinctive Image Features from Scale-Invariant Keypoints [36]. Lowe states that many features from an image will not have any correct match with a training database, due to background clutter or were not detected in the training images. Lowe continues, saying it would be useful to have a way to discard features that do not have a match to the database. He suggests an effective measure to achieve this by comparing the distance of the closest neighbour to that of the second-closest neighbour. Lowe claims, that this measure performs well because correct matches need to have the closest neighbor significantly closer than the closest incorrect match to achieve reliable matching. For false matches, there will likely be a number of other false matches within similar distances due to the high dimensionality of the feature space. Lowe rejects all matches with a distance ratio greater than 0.8, and claims it eliminates 90% of false matches while disregarding 5% of correct matches. The OpenCV documentation specifies a distance ratio of 0.7, therefore my implementation follows suit.

When good matches (matches that have passed Lowe's test) have been filtered, they are placed into the good array. The number of matches in this array are then checked against the minimum matches training parameter. If the length of the array satisfies the minimum matches parameter, then it advances to be checked by the maximum distance parameter. This will take the 20 best matches and take the average distance. If this parameter is satisfied then a positive match is returned.

4.6 The Linguistics Module

In this section I'll give an overview into the implementation of the linguistics module. The systems linguistic module is responsible for processing user input, reasoning with it and producing CNL outputs. This module interacts closely with the communication module, and makes many resource requests to the local CENode agent. It also interacts with the visual module when data from the environment is required.

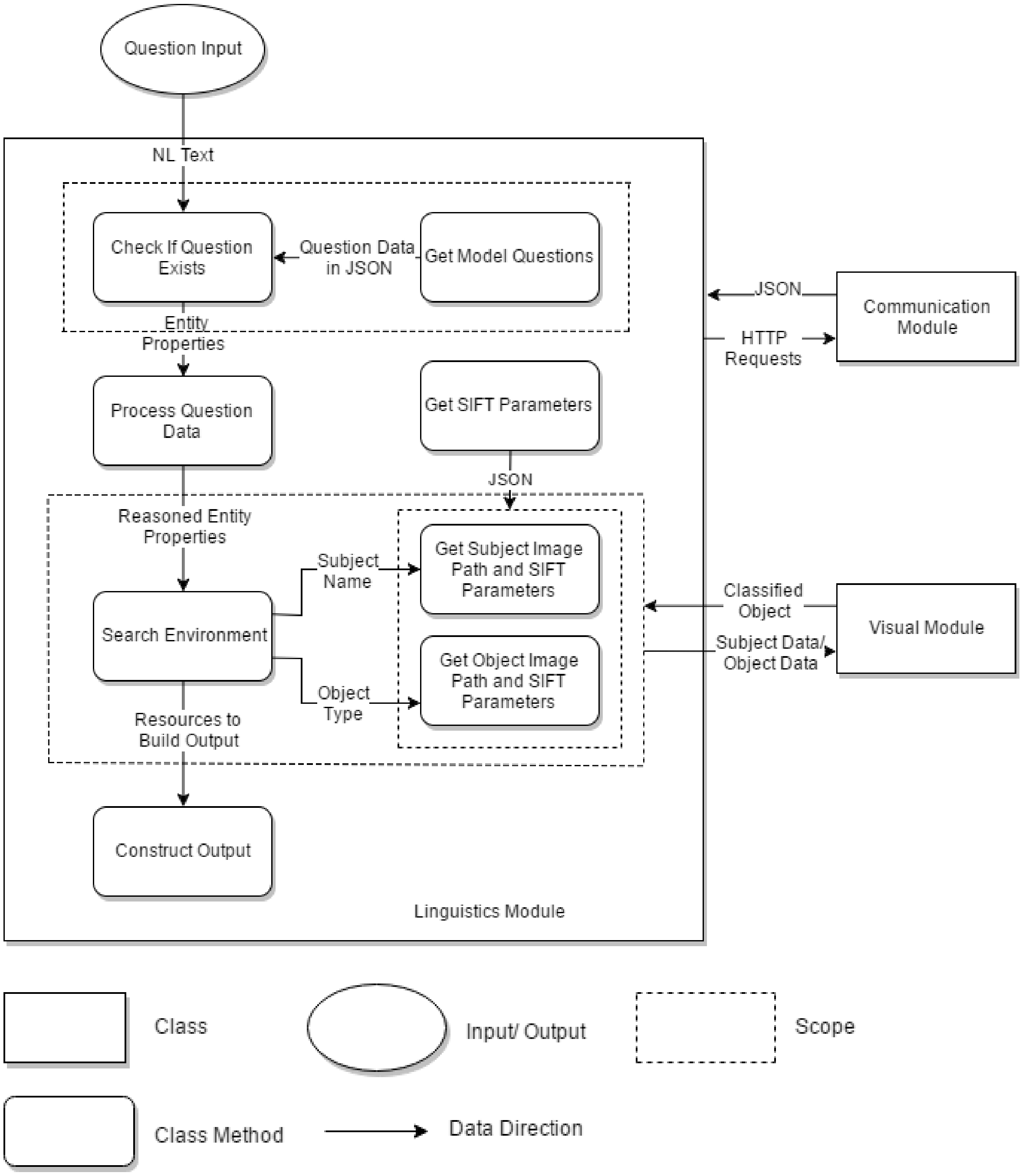
4.61 The Linguistics Class

The linguistics class is initialised with the following constructor.

```
class Linguistic_Module:

    def __init__(self, chat_id):

        self.sherlock_model = Communincation_Module()
        self.instance_data = self.sherlock_model.get_instances()
        self.concept_data = self.sherlock_model.get_concepts()
        self.sift_parameters = self.get_sift_parameters()
```

This constructor will initialise the communication module as well as load instance data, concept data and sift parameters into cache. These datasets are frequently accessed by the class methods, meaning caching them would speed up overall access.

An overview of the linguistics class can be illustrated by the following diagram.



4.62 Check Question Existence

The linguistics class takes a textual input in the form of a question. This will be fed to the `check_if_questions_exist` method, where the question is checked against known questions in the model. This method will retrieve its question data from the `get_model_questions` method, which queries each question instance for the matching question string. As we are not solving complex natural language problems, this solution was deemed acceptable to achieve the goal of the functionality. This solution does have its disadvantages however. As queryable data is limited to the questions in the model, knowledge about non questionable objects such as the ghost, balloon, robot and Gorilla cannot be added to the knowledge base. These objects will only be brought to the systems attention if there is a specific question regarding them. The system has been trained to spot these objects, so a future version of the AI player has the scope to solve this issue.

Given the question ‘What character plays rugby?’, `get_model_questions` will return the following JSON dataset.

```
{"name":"q19","conceptName":"question","conceptId":29,"ce":"there is a question named
'q19' that has 'What character plays rugby?' as text and has 'is played by' as
relationship and concerns the sport
'rugby'.","synonyms":[],"subConcepts":[],"values":[{"label":"text","targetName":"What
character plays rugby?"},{"label":"relationship","targetName":"is played
by"}],"relationships":[{"label":"concerns","targetName":"rugby","targetId":37,"targetCon
ceptName":"sport","targetConceptId":26}]}
```

The code that completes this is as follows;

```
   def check_if_question_exists(self, input_string):
        """ Take question and check if it exists in the model. If it exists, extract
             useful data. """
        input_string = input_string.lower()
        question_data = self.get_model_questions()
        found = False
        for question in question_data:
            if str(question["values"][0]["targetName"]).lower() == input_string:
                subjectType = question["relationships"][0]["targetConceptName"]
                subjectName = question["relationships"][0]["targetName"]
                relationship = question["values"][1]["targetName"]
                questionID = question["name"]
                found = True

        if found:
            self.process_question_data(subjectType, subjectName, relationship,
questionID)
        else:
            print "Invalid Question"
            self.bot.sendMessage(self.chat_id, "Invalid Question")
            return


    def get_model_questions(self):
        """ gets the questions associated with the sherlock game. """
        question_data = []
        for instance in self.instance_data:
            if instance["conceptName"] == "question":

question_data.append(self.sherlock_model.get_model_instance_with_id(str(instance["id"]))
)

        return question_data
```

Observing the `check_if_question_exists` method, we can see how a valid question is found by comparing the `["values"][0]["targetName"]` JSON location with the question contained in `input_string`. If this statement is true, then the subjectType, subjectName, relationship and questionID will be extracted from the dataset. The `found` flag will also be set to True to indicate the question exists, and that the system can proceed to process the extracted data.

In our JSON dataset accompanying the ‘What character plays rugby?’ question, the extracted data would be as follows;

`subjectType = ‘sport’`

`subjectName = ‘rugby’`

`relationship = ‘is played by’`

`questionID = ‘q19’`

4.63 Process Question Data

These values will then passed to the `process_question_data` method, with the aim of finding the object type. To find the object type, the system will traverse the model with its known resources (subjectType, subjectName, relationship) unstil it’s found the correct object type.

This was quite a cumbersome task overall, as numerous data sets had to be iterated through to arrive at the correct object type. This was complicated by certain question types having their object type in different areas of the model. To address this, there are 4 different cases for finding the correct object type, which are dependent on the questionID passed in from the previous method. Some cases have to traverse concepts several times when attempting to come to a conclusion. This consequently results in an increase of data requests to the communication module. In a future implementation, attempts will be made to condense the `process_question_data` method and decrease the number of external resource calls. The code for all 4 cases can be found in Appendix 4.

The most common case for finding the object type can be found below.

```
           # All other questions
            else:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        subjectTypeID = concepts["id"]

                subjectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(subjectTypeID)

                for conceptValues in subjectTypeConcept["relationships"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        break
```

This case will firstly find the `subjectType` within the game concepts. Once a concept has been matched, its ID is extracted. The `get_model_concept_with_id` method is then called with the extracted ID as input. This will return the specified concepts JSON dataset. Given our `subjectType` of `‘Sport’`, its concept ID will passed to the `get_model_concept_with_id` method, with the following JSON dataset returned.

```
{u'relationships': [{u'targetId': 27, u'targetName': u'character', u'label': u'is played
by'}, {u'targetId': 24, u'targetName': u'room', u'label': u'is in'}, {u'targetId': 27,
u'targetName': u'character', u'label': u'is played by'}, {u'targetId': 24,
u'targetName': u'room', u'label': u'is in'}, {u'targetId': 27, u'targetName':
u'character', u'label': u'is played by'}, {u'targetId': 24, u'targetName': u'room',
u'label': u'is in'}, {u'targetId': 27, u'targetName': u'character', u'label': u'is
played by'}, {u'targetId': 24, u'targetName': u'room', u'label': u'is in'}], u'name':
u'sport', u'ce': u'conceptualise a ~ sport ~ S that is a sherlock thing and ~ is played
by ~ the character A and ~ is in ~ the room B and ~ is played by ~ the character C and ~
is in ~ the room D and ~ is played by ~ the character E and ~ is in ~ the room F and ~
is played by ~ the character G and ~ is in ~ the room H.', u'instances': [{u'name':
u'tennis', u'id': 36}, {u'name': u'rugby', u'id': 37}, {u'name': u'soccer', u'id': 38},
{u'name': u'cricket', u'id': 39}, {u'name': u'golf', u'id': 40}, {u'name':
u'basketball', u'id': 41}], u'values': [], u'parents': [{u'name': u'sherlock thing',
u'id': 22}], u'children': []}
```

Iterating through the `'relationships'` key, we're now searching for the `‘is played by’` relationship in its `'label'` subkey. If this relationship is found, then its accompanying `'targetName'` is returned. In our example case, the `'targetName'` value ‘Character’ is found and set as our `objectType`.

Finally the method will output a string informing the user about the objects it'll be searching for in the environment. In our example the following string will be outputted.

We're looking for 'character' associated with 'rugby'

4.64 Searching the Environment

By now the linguistics module would have figured out the subject name, subject type, object type and the relationship between both objects. The missing piece of information is now the object name.

The method `search_environment` is where the linguistics module interacts with the visual module. This interaction has the aim of to resolving the object name and to complete the resources needed to concatenate a CNL string. Using both the subject name and object type, the `search_environment` method will retrieve the appropriate SIFT parameter training sets ready for classification. This can be achieved by calling the `get_subjectName_image_path_and_params` and `get_object_image_paths_and_params` methods.

The `get_subjectName_image_path_and_params` will take a subject name as an input and return the subject name image path, maximum distance, minimum matches and image name in a tuple variable. It is called with the following statement;

```
        subject_data = self.get_subjectName_image_path_and_params(subjectName)
```

and will interact with the following method;

```
    def get_subjectName_image_path_and_params(self, subjectName):
        """ Gets the subjectName image path and get training parameters
            for the image"""
```

```
for index_entry in self.instance_data:
    if index_entry["name"] == subjectName:
        objectID = index_entry["id"]


instance_id_data = self.sherlock_model.get_model_instance_with_id(objectID)

subject_image_path = str(instance_id_data["values"][0]["targetName"])
subject_image_name = str(instance_id_data["name"])

for sift_obj in self.sift_parameters["instances"]:
    if str(sift_obj["name"]).lower() == subjectName.lower() + " parameters":
        siftID = sift_obj["id"]

instance_id_data = self.sherlock_model.get_model_instance_with_id(siftID)

for value in instance_id_data["values"]:
    if value["label"] == "maximum distance":
        max_distance = value["targetName"]
    elif value["label"] == "minimum matches":
        min_matches = value["targetName"]

parameters = (subject_image_path, max_distance, min_matches, subject_image_name)
return parameters
```

This method will initially find the subject name's ID from the instance data cache, and call `get_model_instance_with_id` with the ID input. Assuming our subject name is 'Rugby' we'd receive the following JSON response.

```
{u'relationships': [], u'conceptId': 26, u'name': u'rugby', u'conceptName': u'sport',
u'ce': u"there is a sport named 'rugby' that has 'rugby.jpg' as image.", u'synonyms':
[], u'values': [{u'targetName': u'rugby.jpg', u'label': u'image'}], u'subConcepts': []}
```

From this dataset we can extract the subject's training image path by accessing the `instance_id_data["values"][0]["targetName"]` JSON location. The next step is to query the SIFT parameters associated with the training image. Assuming our SIFT parameter set is called 'rugby parameters', we would retrieve its instance ID from cache and query the `get_model_instance_with_id` with it. This would return the following JSON response.

```
{u'relationships': [], u'conceptId': 30, u'name': u'rugby parameters', u'conceptName':
u'sift parameter set', u'ce': u"there is a sift parameter set named 'rugby parameters'
that has the sport 'rugby' as related thing and has '30' as maximum distance and has
'125' as minimum matches.", u'synonyms': [], u'values': [{u'targetConceptId': 26,
u'targetId': 37, u'targetName': u'rugby', u'targetConceptName': u'sport', u'label':
u'related thing'}, {u'targetName': u'30', u'label': u'maximum distance'},
{u'targetName': u'125', u'label': u'minimum matches'}], u'subConcepts': []}
```

Iterating through the `'values'` key, the strings 'maximum distance' and 'minimum matches' must be matched to their `'label'` subkey. The values will then be available in the accompanying `'targetName'` key, and set to the max_distance and min_matches variable.

This process was repeated for the `get_object_image_paths_and_params` method with an input of object type. With the object type in our example being 'Character', this method would return the SIFT parameters relating to all 6 characters within the game. With both parameter sets now determined, the linguistics module can now prompt the visual module to find the missing object name. The following code represents this call.

```
image_recognition = Visual_Module(self.chat_id)
found_object = image_recognition.find_scenes_with_objects(subject_data, object_data)
```

Our example will return the character ‘Giraffe’ and store it in the `found_object` variable.

4.65 Constructing Output

Our system would now have all the resources it needs to construct a CNL string containing an answer to the question. This CNL string will be constructed and outputted in the `construct_output` method, followed by its insertion into a tell card.

Taking the inputs subjectType, subjectName, relationship, objectType and found_object, the CNL string is concatenated with the following code;

```
message_to_output = "The {0} \\'{1}\\' {2} the {3} \\'{4}\\'".format(subjectType,
subjectName, relationship, objectType, found_object)

message_to_print = "The {0} '{1}' {2} the {3} '{4}'".format(subjectType, subjectName,
relationship, objectType, found_object)
```

The `message_to_output` variable is concatenated into a tell card, whereas `message_to_print` is outputted to standard out. The `message_to_output` string contains escaped symbols ‘\\’, to ensure the single quotes aren’t misinterpreted when integrated with the tell card string and by extension to the central CENode agent.

For our question example, the two following strings will be produced.

```
message_to_output = The sport \\'rugby\\' is played by the character \\'Giraffe\\'
message_to_print = The sport 'rugby' is played by the character 'Giraffe'
```

The final step in our implementation is to construct the tell card and to post it to the shared knowledge base.

Cards are defined in CENode as a delivery mechanism for CE, and are recommended as primary means for human-node and node-node communication. Cards wrap CE contain a value property and enable the information within to be shipped to different agents. This system will be using a subclass of the card concept known as a tell card to communicate its observations. Tell cards act as a vehicle to post valid CE to a node, and they’re the only card whose contents can modify a knowledge base.

Our example will produce the following tell card.

```
there is a tell card named 'msg_{uid}' that has 'The sport \'rugby\' is played by the
character \'Giraffe\'' as content and is to the agent 'sherlock' and is from the agent
'sherBot' and has the timestamp '{now}' as timestamp
```

With the card ready to be posted, the system calls the communication module and posts the card with the `post_to_shared_kb` method.

```
self.sherlock_model.post_to_shared_kb(tellcard)
```

4.7 SHERLOCK Dashboard

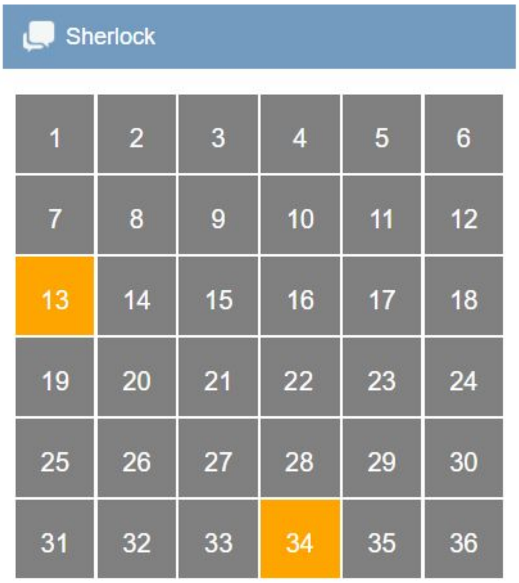
One of the key deliverables of this project is to see the SHERLOCK dashboard ‘change colour’, indicating the central agent has received new knowledge. The dashboard is available at http://rob.cenode.io/ and is listening the `‘sherlock’` agent being hosted on http://explorer.cenode.io:6789. The system posts its tell cards to http://explorer.cenode.io:6789, which in turn saves them as card instances. The tell card in our example case is represented by the following JSON dataset.

```
{"name":"msg_sherlock2","conceptName":"tell card","conceptId":7,"ce":"there is a tell
card named 'msg_sherlock2' that has 'The sport \\'rugby\\' is played by the character
\\'Giraffe\\'' as content and has the timestamp '1493422086125' as timestamp and is to
the agent 'sherlock' and is from the agent 'sherBot' and is to the agent
'Moira'.","synonyms":[],"subConcepts":[],"values":[{"label":"content","targetName":"The
sport 'rugby' is played by the character
'Giraffe'"},{"label":"timestamp","targetName":"1493422086125","targetId":7,"targetConcep
tName":"timestamp","targetConceptId":3}],"relationships":[{"label":"is
to","targetName":"sherlock","targetId":4,"targetConceptName":"agent","targetConceptId":4
},{"label":"is
from","targetName":"sherBot","targetId":6,"targetConceptName":"agent","targetConceptId":
4},{"label":"is
to","targetName":"Moira","targetId":3,"targetConceptName":"agent","targetConceptId":4}]}
```

The SHERLOCK agent operational at http://rob.cenode.io/ is running the same modified SHERLOCK model we’ve been using, however it also contains some extra instructional statements.

```
var SHERLOCK_NODE_MODEL = [
  "there is an agent named 'sherlock' that has 'http://explorer.cenode.io:6789' as
address",
  "there is a tell policy named 'p2' that has 'true' as enabled and has the agent
'sherlock' as target",
  "there is a listen policy named 'p4' that has 'true' as enabled and has the agent
'sherlock' as target"
];
```

These statement inform the dashboard agent about the central agent named ‘sherlock’ at the address http://explorer.cenode.io:6789. It then sets both its tell policy and listen policies to true, along with the ‘sherlock’ agent as a target. It’s these statements that enable http://rob.cenode.io/ to listen for incoming knowledge and to update the dashboard accordingly. The SHERLOCK instance will see our tell card within the shared knowledge base and update its accompanying tile(s). Here’s the dashboard following the addition of our example case tell card.



Sherlock

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

We can see that square 13 and 24 have changed their colours from grey to orange. This indicates that some knowledge has been entered regarding these questions, however there are not enough statements to form a conclusive answer. As human players add knowledge regarding the questions, the tiles will change to green (conclusive) or red (un-conclusive).

Tile 13 represents the question ‘What character plays rugby?’, whereas tile 34 represents ‘What sport does Giraffe play?’. As the CE model is bidirectional, the knowledge we’ve added has provided an answer to both questions.

5.0 Testing

In this section I’ll be testing the system against the requirements and use case outlined in the part 3.2.

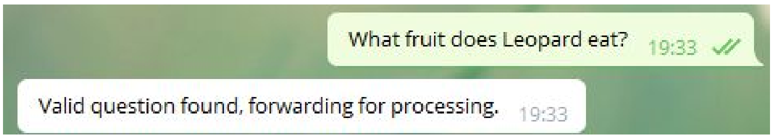
5.1 The system must be prompted to answer a question

Acceptance Criteria

- A question in the form of valid textual input must be received and processed by the AI player.
- Only valid questions in the SHERLOCK game must be acted upon.

The system can be prompted to answer a question by sending a message to the user @cf_sher_bot via the telegram messaging service. Textual input will be forwarded to the system, where its validity will be determined. A valid question is determined as one within the SHERLOCK game. All valid questions can be found in Appendix 6.

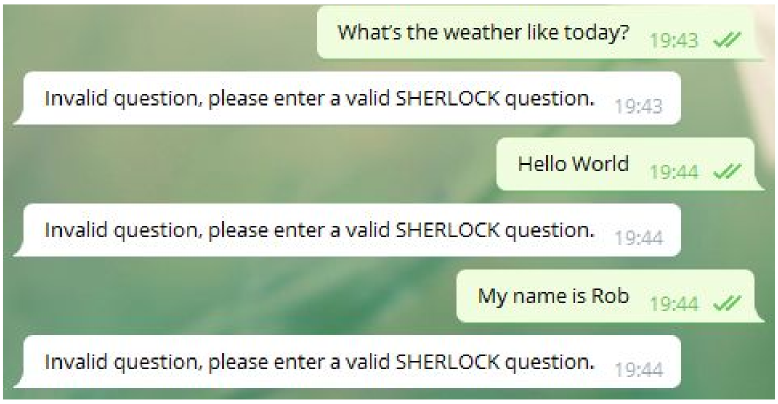
Asking the **valid question** eg: ‘What fruit does Leopard eat?’, will return the following;



What fruit does Leopard eat? 19:33

Valid question found, forwarding for processing. 19:33

Asking an **invalid question** e.g: ‘What’s the weather like today?’, ‘Hello World’, ‘My name is Rob’ will return the following;



What’s the weather like today? 19:43

Invalid question, please enter a valid SHERLOCK question. 19:43

Hello World 19:44

Invalid question, please enter a valid SHERLOCK question. 19:44

My name is Rob 19:44

Invalid question, please enter a valid SHERLOCK question. 19:44

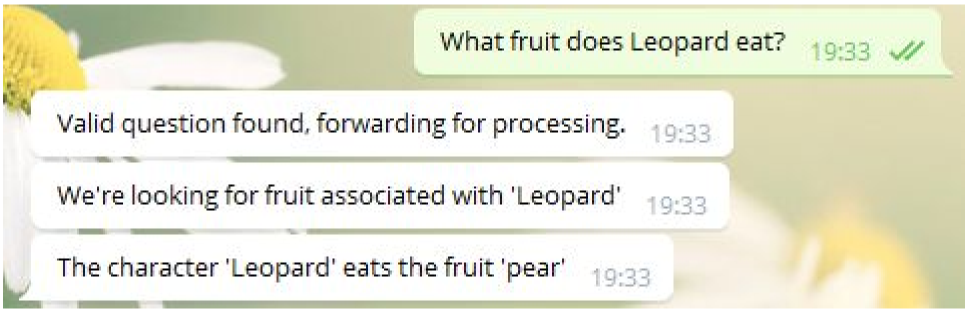
As we can see, the system can be prompted to answer textual input by via the telegram messaging service, while also filtering out invalid input.

5.2 The system must produce CNL output with relationships

Acceptance Criteria

- A CNL output must be constructed in the form of valid ITA CE.
- The CNL output must have the relationship between both objects.

The system must produce a valid CNL string in ITA CE in order to communicate what it’s observed to the central agent. This string must contain two objects, the type of objects and an adjoining relationship between both objects. Following a valid question input, a CNL string with a relation between two objects will be produced. Given the following question “What fruit does Leopard eat”;



What fruit does Leopard eat? 19:33

Valid question found, forwarding for processing. 19:33

We're looking for fruit associated with 'Leopard' 19:33

The character 'Leopard' eats the fruit 'pear' 19:33

The system will produce the CNL string “The character 'Leopard' eats the fruit 'pear'”. The string contains 2 objects (Leopard, Pear), 2 object types (Character, fruit) and a combining relationship (eats).

5.3 The system must play the game alongside human players

Acceptance Criteria

- The system must communicate its perceptions via tell cards to the central agent.
- An empty SHERLOCK dashboard listening to the central agent must change from grey to yellow when a valid tell card is posted.
- A human player must also send an observation to the central agent via the SHERLOCK conversational interface.
- The human player's observation must also change the colour of the SHERLOCK dashboard, with the ai players observation clearly visible.
- The human agent must “ask” its conversational agent a question regarding the AI players observation.

When cards are submitted to the shared knowledge base, the users local agent updates its SHERLOCK dashboard by listening for instances of tell cards in the central agent.

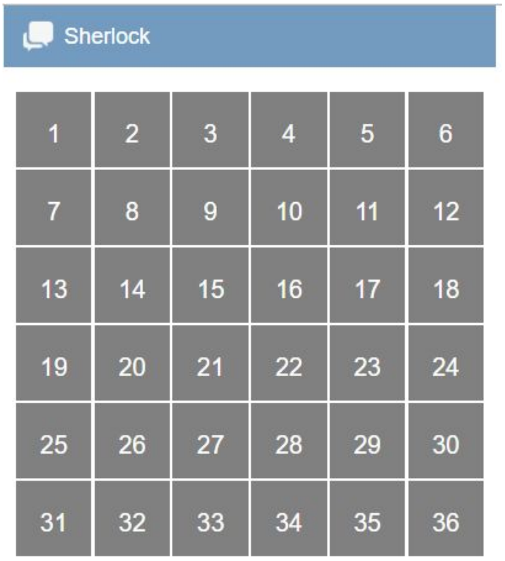
Tell cards are constructed by the linguistics module and posted by the communication module. The following is an example of a tell card;

```
there is a tell card named 'msg_{uid}' that has 'The sport \'rugby\' is played by the
character \'Giraffe\'' as content and is to the agent 'sherlock' and is from the agent
'sherBot' and has the timestamp '{now}' as timestamp
```

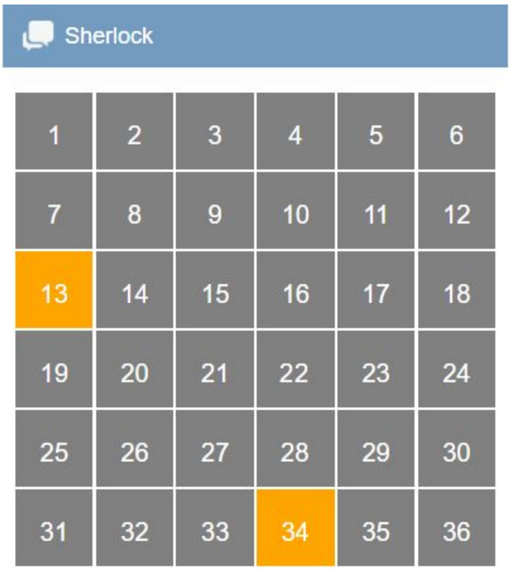
If this card is successfully posted, its instance on the central agent will take the form of the following JSON dataset;

```
{"name":"msg_sherlock2","conceptName":"tell card","conceptId":7,"ce":"there is a tell
card named 'msg_sherlock2' that has 'The sport \\'rugby\\' is played by the character
\\'Giraffe\\'' as content and has the timestamp '1493422086125' as timestamp and is to
the agent 'sherlock' and is from the agent 'sherBot' and is to the agent
'Moira'.","synonyms":[],"subConcepts":[],"values":[{"label":"content","targetName":"The
sport 'rugby' is played by the character
'Giraffe'"},{"label":"timestamp","targetName":"1493422086125","targetId":7,"targetConcep
tName":"timestamp","targetConceptId":3}],"relationships":[{"label":"is
to","targetName":"sherlock","targetId":4,"targetConceptName":"agent","targetConceptId":4
},{"label":"is
from","targetName":"sherBot","targetId":6,"targetConceptName":"agent","targetConceptId":
4},{"label":"is
to","targetName":"Moira","targetId":3,"targetConceptName":"agent","targetConceptId":4}]}
```

The user agents local SHERLOCK dashboard will react to this tell card and update the appropriate squares from grey to yellow. Observing from a human player's perspective we can see the dashboard has changed from being fully grey to having 2 squares contain some knowledge. These represent the tell card observations from the AI player.

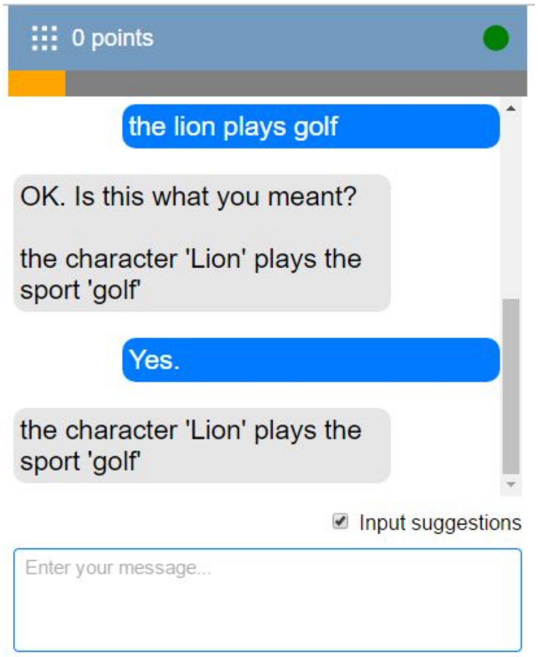


1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36



1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

As the human player we can add our own observations which will update the same dashboard.



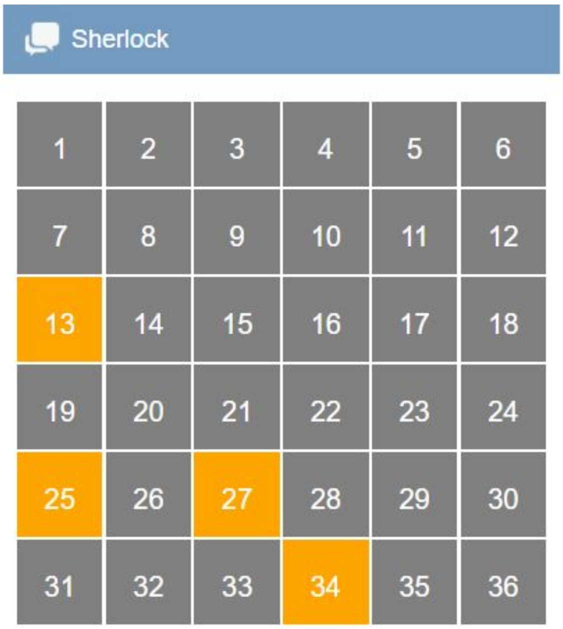
the lion plays golf

OK. Is this what you meant?

the character 'Lion' plays the sport 'golf'

Yes.

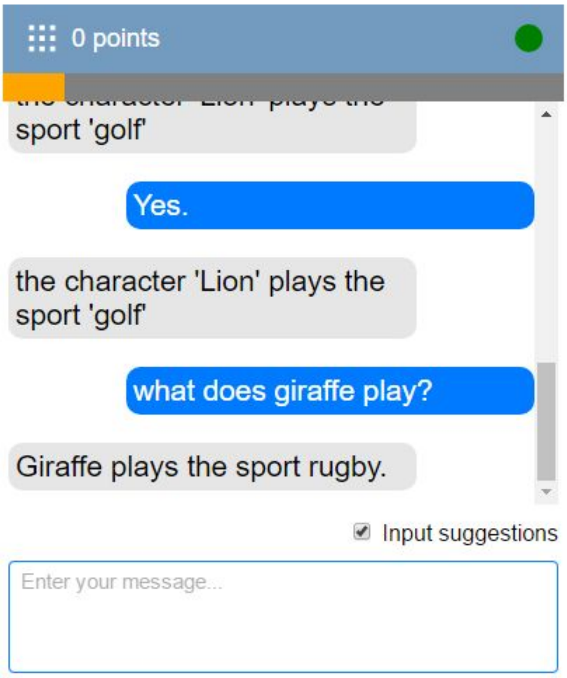
the character 'Lion' plays the sport 'golf'



1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Note: Although 2 observations have been inputted, they've both answered 2 questions each; hence there are 4 squares lit up.

A human player can then perform an ask prompt regarding the AI players answered question to the central agent.



sport 'golf'

Yes.

the character 'Lion' plays the sport 'golf'

what does giraffe play?

Giraffe plays the sport rugby.

This proves that observations from the AI player are being received and communicated by the central node, with other players able access them.

5.4 The system should use the same or similar model of the world as human players.

Acceptance Criteria

- The AI player must have its own local user agent with the a version of the SHERLOCK CE model on it.
- This local user agent must provide the AI player with the same concepts, objects, rules and relationships available to human players.

The communication module is responsible for querying the local host CENode instance for game instances and concepts. It contains 4 different GET calls to achieve this. Accessing this data implies the AI player has access to the same resources a human player agent would have.

Calling `get_concepts` in the communication model will return all concepts;

```
[{"name":"entity","id":1},{"name":"imageable
thing","id":2},{"name":"timestamp","id":3},{"name":"agent","id":4},{"name":"individual",
"id":5},{"name":"card","id":6},{"name":"tell card","id":7},{"name":"ask
card","id":8},{"name":"gist card","id":9},{"name":"nl card","id":10},{"name":"confirm
card","id":11},{"name":"location","id":12},{"name":"locatable
thing","id":13},{"name":"rule","id":14},{"name":"policy","id":15},{"name":"tell
policy","id":16},{"name":"ask policy","id":17},{"name":"listen
policy","id":18},{"name":"listen onbehalfof policy","id":19},{"name":"forwardall
policy","id":20},{"name":"feedback policy","id":21},{"name":"sherlock
thing","id":22},{"name":"fruit","id":23},{"name":"room","id":24},{"name":"hat
colour","id":25},{"name":"sport","id":26},{"name":"character","id":27},{"name":"object",
"id":28},{"name":"question","id":29},{"name":"sift parameter set","id":30}]
```

Calling `get_instances` in the communication module will return all instances;

See Appendix 3

Calling `get_model_concept_with_id` with an id argument will return a specific concepts attributes. (eg: `{"name":"location","id":12}`)

```
{"name":"location","ce":"conceptualise a ~ location ~ L that is a
entity.","parents":[{"name":"entity","id":1}],"children":[{"name":"room","id":24}],"inst
ances":[],"values":[],"relationships":[]}
```

Calling `get_model_instances_with_id` with an id argument will return a specific instances attributes. (eg: `{"name":"Zebra","id":17}`)

```
{"name":"Zebra","conceptName":"character","conceptId":27,"ce":"there is a character
named 'Zebra' that has 'zebra.jpg' as
image.","synonyms":[],"subConcepts":[],"values":[{"label":"image","targetName":"zebra.jp
g"}],"relationships":[]}
```

5.5 The system must not launch if a local CENode instance is not operational

Acceptance Criteria

- An error must appear informing that a local CENode instance is not operational, followed by the system exiting successfully.

The system can be launched by executing the `ai_player.py` file in a terminal window from our working directory.

```
python ai_player.py
```

If the local CENode instance is not operation, the following error message will be thrown.

```
(cv) rob@ROB-PC:/mnt/c/Users/JME/git/scene_recognition$ python ai_player.py

Error posting model to instance. Please launch the localhost CEServer to continue.
HTTPConnectionPool(host='localhost', port=8004): Max retries exceeded with url:
/sentences (Caused by
NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at
0x7fb88b64d650>: Failed to establish a new connection: [Errno 111] Connection
refused',))

(cv) rob@ROB-PC:/mnt/c/Users/JME/git/scene_recognition$
```

The message above throws a requests API exception informing the user of a failed attempt in contacting a URL. The error message informs the user to launch the localhost CEServer to continue. The system has exited successfully as no stack traceback has been thrown, and control has been returned to the terminal.

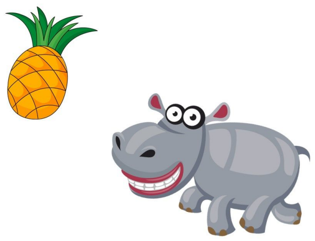
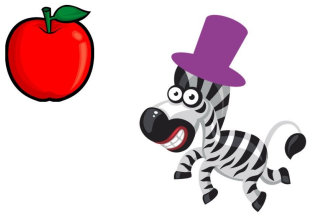
Alternatively, a successful launch would look like the following;

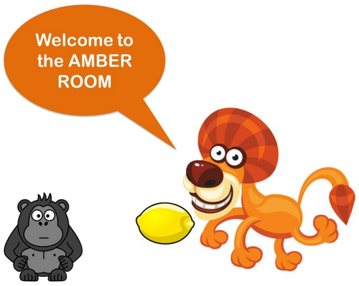
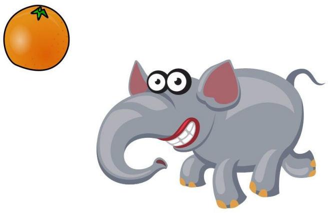
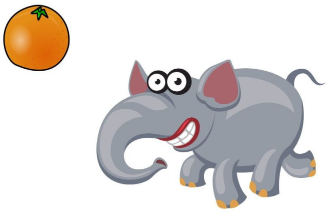
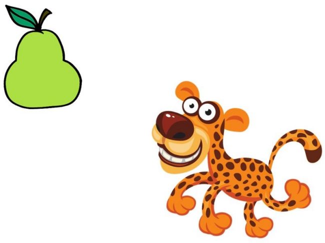
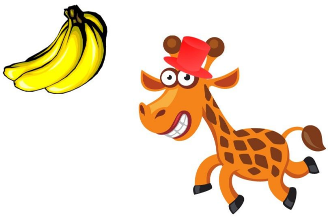
```
(cv) rob@ROB-PC:/mnt/c/Users/JME/git/scene_recognition$ python ai_player.py
AI Player Launched!
```

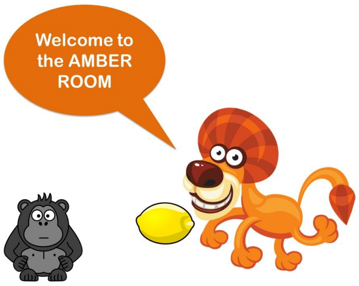
–

5.6 The AI player must use computer vision techniques to accurately perceive objects in a scene

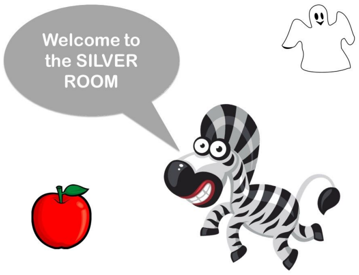
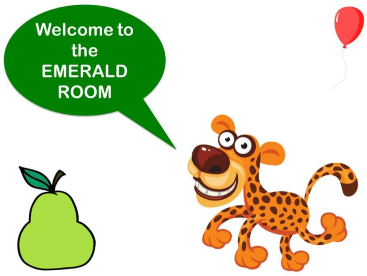
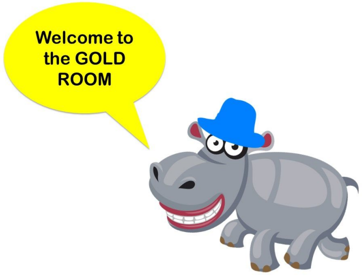
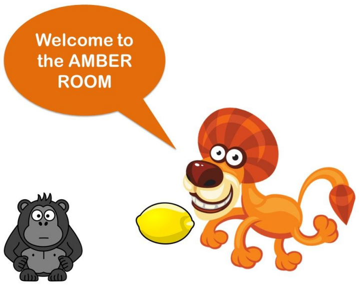
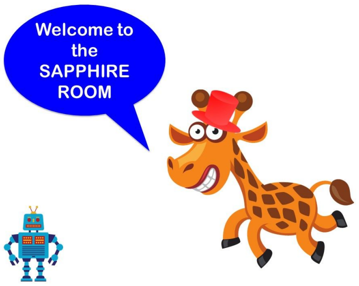
It is vital that knowledge produced by the AI player is accurate or we'd risk polluting the shared knowledge base with inaccurate knowledge. Producing accurate observations would also support the viability of an artificial system in contributing to crowdsourced datasets. To test the accuracy of the system I'll produce table containing the questions, CNL string produced from the questions, a scene relating to the question and an accuracy indicator.

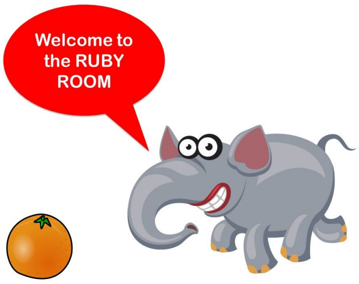
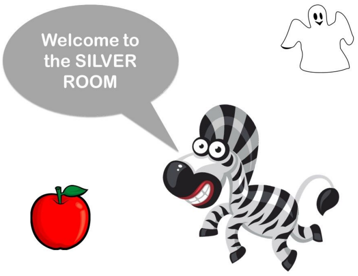
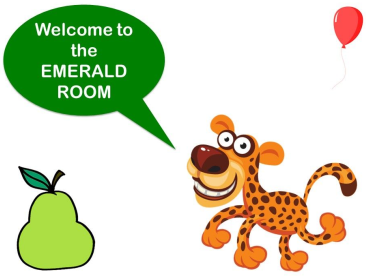
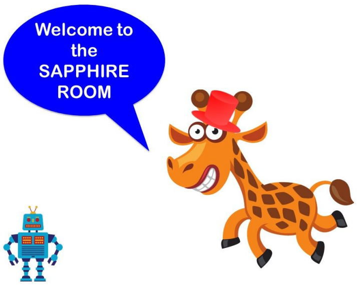
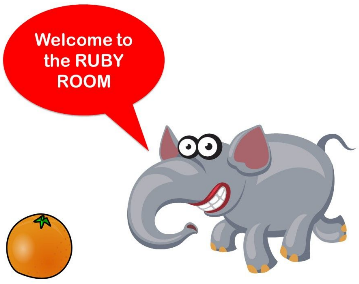
Question	CNL Output	Scene Relating to Question	Correct Observation
What character eats pineapples?	The fruit 'pineapple' is eaten by the character 'Hippopotamus'		Yes
What character eats apples?	The fruit 'apple' is eaten by the character 'Zebra'		Yes
What character eats bananas?	The fruit 'banana' is eaten by the character 'Giraffe'		Yes

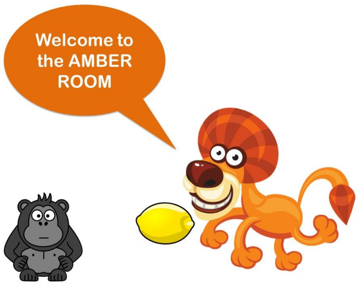
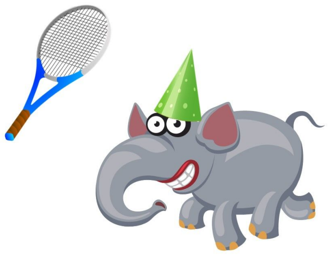
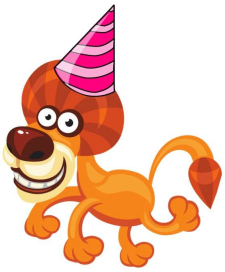
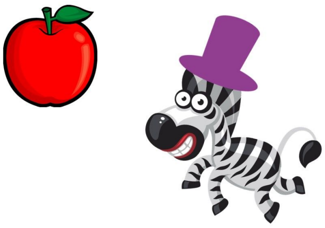
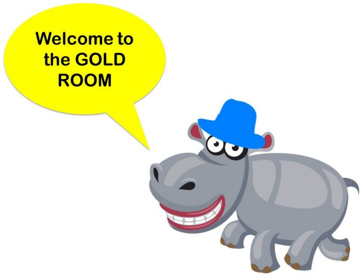
What character eats lemons?	The fruit 'lemon' is eaten by the character 'Lion'		Yes
What character eats oranges?	The fruit 'orange' is eaten by the character 'Elephant'		Yes
What fruit does Elephant eat?	The character 'Elephant' eats the fruit 'orange'		Yes
What fruit does Leopard eat?	The character 'Leopard' eats the fruit 'pear'		Yes
What fruit does Giraffe eat?	The character 'Giraffe' eats the fruit 'banana'		Yes

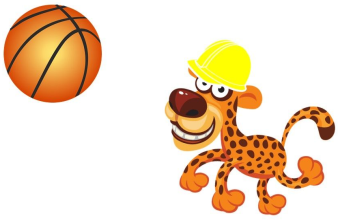
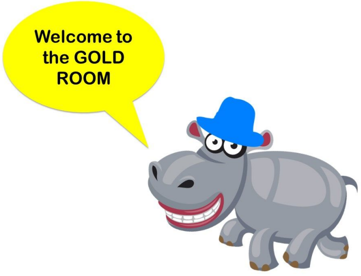
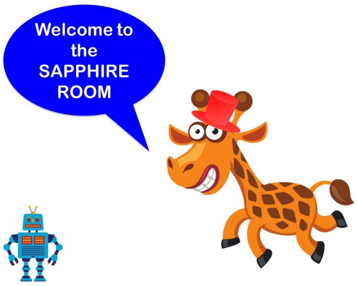
What fruit does Lion eat?	The character 'Lion' eats the fruit 'lemon'		Yes
What sport does Zebra play?	The character 'Zebra' plays the sport 'cricket'		Yes
What sport does Lion play?	The character 'Lion' plays the sport 'golf'		Yes
What sport does Giraffe play?	The character 'Giraffe' plays the sport 'rugby'		Yes
What sport does Hippopotamus play?	The character 'Hippopotamus' plays the sport 'soccer'		Yes

What sport does Elephant play?	The character 'Elephant' plays the sport 'tennis'		Yes
What character plays rugby?	The sport 'rugby' is played by the character 'Giraffe'		Yes
What character plays basketball?	The sport 'basketball' is played by the character 'Leopard'		Yes
What character plays soccer?	The sport 'soccer' is played by the character 'Hippopotamus'		Yes
What character plays golf?	The sport 'golf' is played by the character 'Lion'		Yes

Where is the apple?	The fruit 'apple' is in the room 'Silver Room'		Yes
Where is the pear?	The fruit 'pear' is in the room 'Emerald Room'		Yes
Where is Hippopotamus?	The character 'Hippopotamus' is in the room 'Gold Room'		Yes
Where is Lion?	The character 'Lion' is in the room 'Amber Room'		Yes
Where is Giraffe?	The character 'Giraffe' is in the room 'Sapphire Room'		Yes

Where is Elephant?	The character 'Elephant' is in the location 'Ruby Room'		Yes
What fruit is in the silver room?	The fruit 'apple' is in the room 'Silver Room'		Yes
Which character is in the emerald room?	The character 'Leopard' is in the room 'Emerald Room'		Yes
What character is in the sapphire room?	The character 'Giraffe' is in the room 'Sapphire Room'		Yes
What character is in the ruby room?	The character 'Elephant' is in the room 'Ruby Room'		Yes

What character is in the amber room?	The character 'Lion' is in the room 'Amber Room'		Yes
What colour hat is Elephant wearing?	The hat colour 'green' is worn by the character 'Elephant'		Yes
What colour hat is Lion wearing?	The hat colour 'pink' is worn by the character 'Lion'		Yes
What colour hat is Zebra wearing?	The hat colour 'purple' is worn by the character 'Zebra'		Yes
What colour hat is Hippopotamus wearing?	The hat colour 'blue' is worn by the character 'Hippopotamus'		Yes

What character is wearing a yellow hat?	The hat colour 'yellow' is worn by the character 'Leopard'		Yes
What character is wearing a blue hat?	The hat colour 'blue' is worn by the character 'Hippopotamus'		Yes
What character is wearing a red hat?	The hat colour 'red' is worn by the character 'Giraffe'		Yes

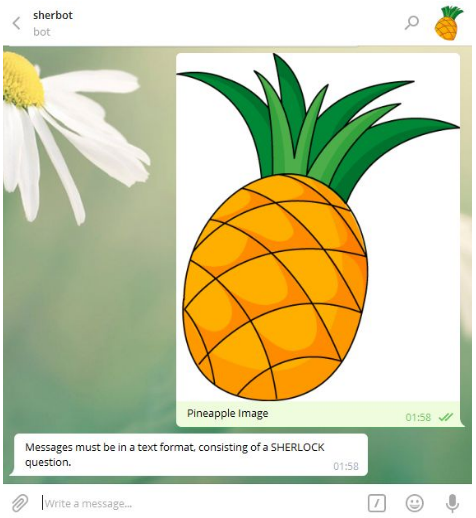
The system has a 100% success rate when classifying objects using the computer vision solution inside the visual module.

5.7 Invalid input must be handled gracefully

Acceptance Criteria

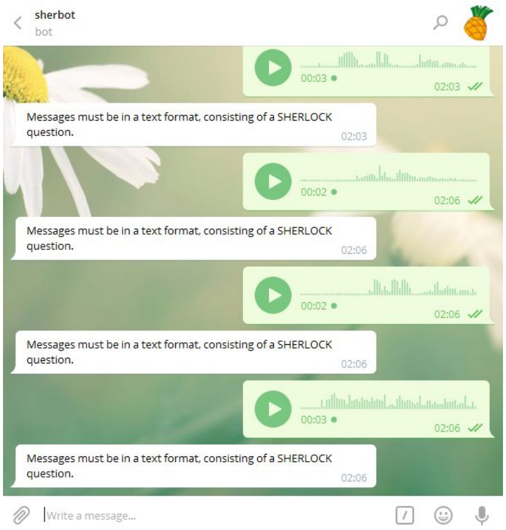
- The system should catch invalid input and not proceed until valid input is given.
- Invalid input must return an informative error, specifying what type of input the system requires.

Using the telegram as a message relay it is possible to send non textual messages to the system (eg: Images/ file attachments). The system must be able to handle and reject these non textual inputs and provide an informative error message.



Pineapple Image 01:58

Messages must be in a text format, consisting of a SHERLOCK question. 01:58



Messages must be in a text format, consisting of a SHERLOCK question. 02:03

Messages must be in a text format, consisting of a SHERLOCK question. 02:06

Messages must be in a text format, consisting of a SHERLOCK question. 02:06

Messages must be in a text format, consisting of a SHERLOCK question. 02:06

The above images represent the system's response to an image input and a voice message input. The error message reads;

Messages must be in a text format, consisting of a SHERLOCK question.

5.8 Unsuccessful HTTP requests must be handled gracefully

Acceptance Criteria

- If a HTTP error is encountered, it must be caught and returned with an informative message on which endpoint is not responding. eg (If a localhost agent instance is not active)

The system performs several HTTP requests for external sources while reasoning with game questions. It is important that unexpected connection issues are handled sensibly and gracefully so the issue can be solved swiftly. Each method in the communication module contains its own unique error message and requests API exception.

`post_model` error message;

```
Error posting model to instance. Please check your connection with the localhost CEServer.

HTTPConnectionPool(host='localhost', port=8004): Max retries exceeded with url: /sentences
(Caused by NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object
at 0x7fb88b64d650>: Failed to establish a new connection: [Errno 111] Connection
refused',))
```

`post_to_shared_kb` error message;

```
Posting to shared knowledge base failed. Please check your connection with the external
instance at http://explorer.cenode.io.

HTTPConnectionPool(host='explorer.cenode.io', port=6789): Max retries exceeded with url:
/sentences (Caused by
NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at
0x7fb9ed696350>: Failed to establish a new connection: [Errno 111] Connection
refused',))
```

`get_instances` error message;

```
Unable to get instances. Please check your connection with the localhost CEServer.

HTTPConnectionPool(host='localhost', port=8004): Max retries exceeded with url:
/instances (Caused by
NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at
0x7f79ab2cded0>: Failed to establish a new connection: [Errno 111] Connection
refused',))
```

`get_concepts` error message;

```
Unable to get concepts. Please check your connection with the localhost CEServer.

HTTPConnectionPool(host='localhost', port=8004): Max retries exceeded with url:
/concepts (Caused by
NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at
0x7f68876bf150>: Failed to establish a new connection: [Errno 111] Connection
refused',))
```

`get_model_concept_with_id` error message;

```
Unable to get concept with ID. Please check the concept ID exists and your connection
with the localhost CENode exists.

HTTPConnectionPool(host='localhost', port=8004): Max retries exceeded with url:
/concept?id=30 (Caused by
NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at
0x7ff0a7d7e250>: Failed to establish a new connection: [Errno 111] Connection
refused',))
```

`get_model_instance_with_id` error message;

```
Unable to get instance ID. Please check the instance ID exists and your connection with
the localhost CENode exists.

HTTPConnectionPool(host='localhost', port=8004): Max retries exceeded with url:
/instance?id=47 (Caused by
NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at
0x7fe460b7e590>: Failed to establish a new connection: [Errno 111] Connection
refused',))
```

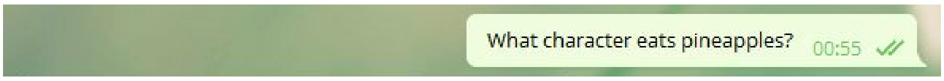
5.9 Use Case

User prompts the system with a question

Basic Flow

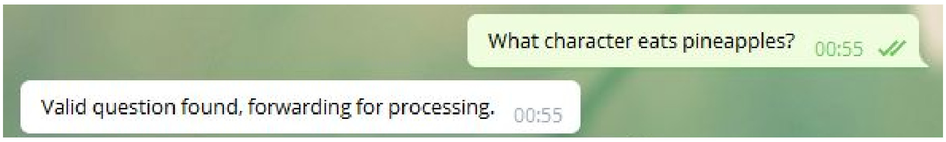
1. User prompts the system with a valid SHERLOCK question.

Prompting the system with the question “What character eats pineapples?”



What character eats pineapples?

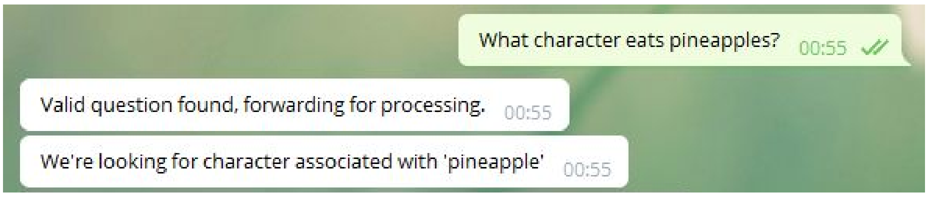
2. System acknowledges the question and reasons with it.



What character eats pineapples? 00:55

Valid question found, forwarding for processing. 00:55

3. System scans it's poster "feeds" with inputs derived from the question.

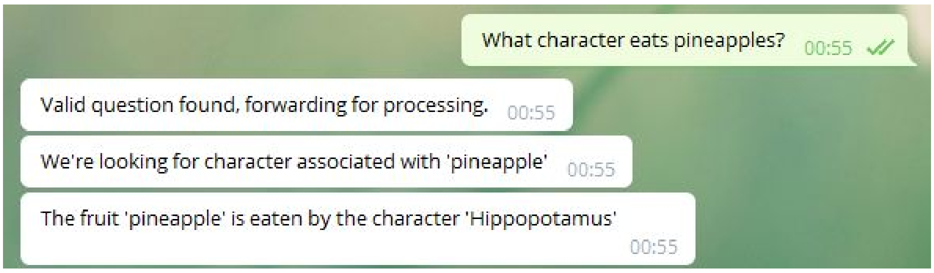


What character eats pineapples? 00:55

Valid question found, forwarding for processing. 00:55

We're looking for character associated with 'pineapple' 00:55

4. System returns its observation in CNL to the user.



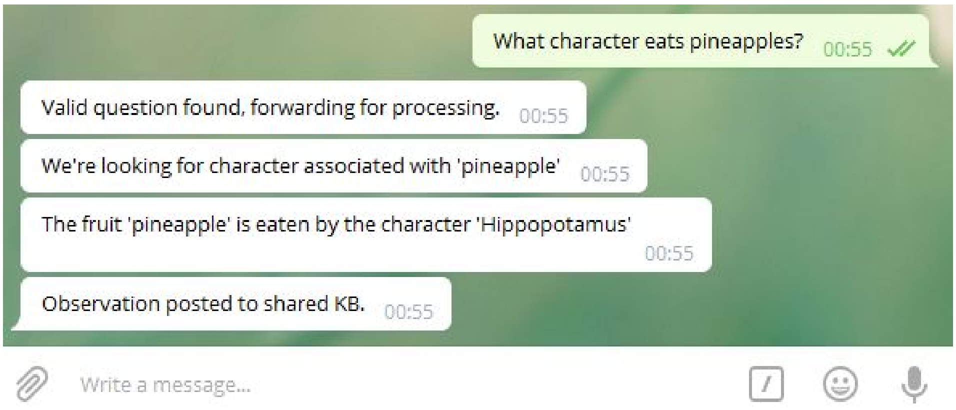
What character eats pineapples? 00:55

Valid question found, forwarding for processing. 00:55

We're looking for character associated with 'pineapple' 00:55

The fruit 'pineapple' is eaten by the character 'Hippopotamus' 00:55

5. System sends its observation to the shared KB.



What character eats pineapples? 00:55

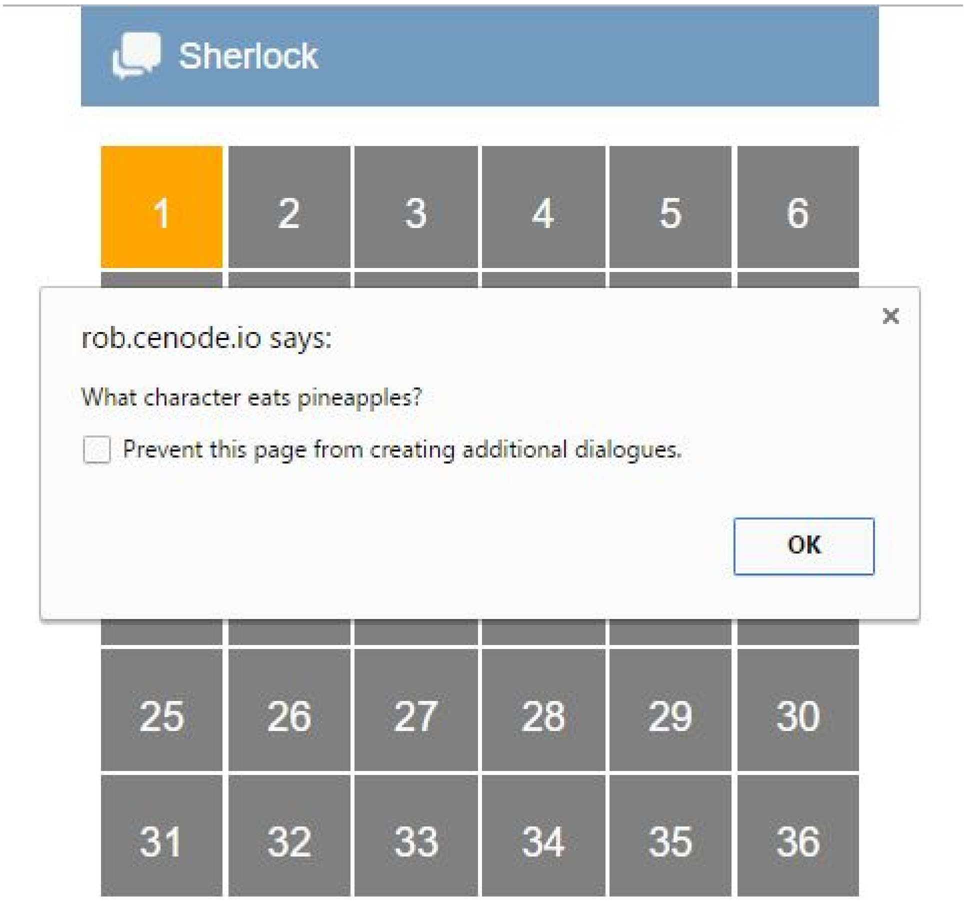
Valid question found, forwarding for processing. 00:55

We're looking for character associated with 'pineapple' 00:55

The fruit 'pineapple' is eaten by the character 'Hippopotamus' 00:55

Observation posted to shared KB. 00:55

The dashboard acknowledges the posted observation;



rob.cenode.io says:

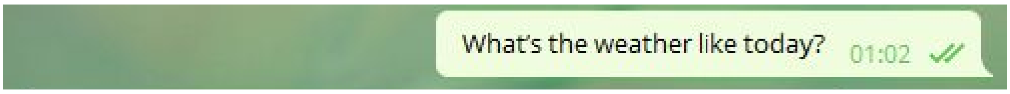
What character eats pineapples?

☐ Prevent this page from creating additional dialogues.

Alternative flow

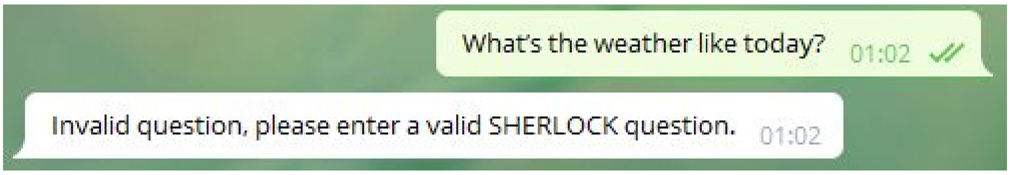
Prompting the system with invalid input “What’s the weather like today?”

1. User prompts the system with invalid input.



What’s the weather like today? 01:02

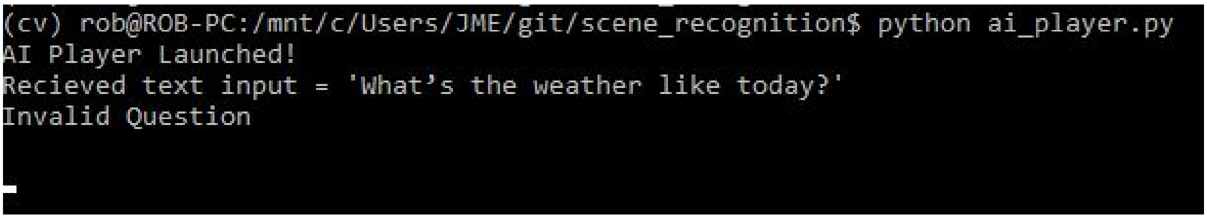
2. System rejects the input and raises and error.



What’s the weather like today? 01:02

Invalid question, please enter a valid SHERLOCK question. 01:02

3. System returns to a “listening” state where it’s waiting for input.



```
(cv) rob@ROB-PC:/mnt/c/Users/JME/git/scene_recognition$ python ai_player.py
AI Player Launched!
Recieved text input = 'What’s the weather like today?'
Invalid Question
```

The “_” icon in the terminal window indicates the system is “listening” for input.

6.0 Future work

There is much scope for future work based off this project, especially in expanding system to be compatible with different problem domains. Different problem domains could entail different environments requiring different cognitive abilities such as sound and sensing (temperature, light, chemical). The problem domain could also extend to non english controlled language models, such as controlled natural welsh. Perceiving the world in Welsh, the node could produce CNW output, and report Welsh observations to a central agent.

Although the system currently possesses many cognitive architecture attributes, it’s declarative long term memory is currently missing. In the context of this project, the declarative memory would be the AI players local agent maintaining its knowledge base with its own observations, and receive external observations from the central node. The AI players current implementation does not maintain its local agent, and can only communicate what it observes with the central agent. Future work would focus on enabling this bilateral communication between the AI players local agent and the central agent. Once it’s implemented the AI players local agent can receive up to date knowledge on the state of the environment from other players, as well as post its own observations.

The initial purpose of the SHERLOCK game was to research human behaviour, while interacting with a CNL agent. Provided with limited information about the initial world, humans are tasked with collecting information they have observed locally. Using the CNL agent, humans must collect additional information about the world by working as a team. Experimenting in a controlled environment, observations about human interaction and the quality of collected information are made. It would be interesting having the AI player operate alongside the human players while one of these experiments is taking place. I suspect the AI player may have the biggest impact on the mean score, relating to the certainty of answered questions. This is because the AI player has shown 100% accuracy so far in observing its environment. The AI player can operate by being prompted or could be configured to run autonomously in the background. The AI player could perform all of its observations instantly, or it can be spaced to intervals in the game (eg: every 30 seconds). Once the players declarative memory is working correctly it could behave dynamically, searching for gaps in knowledge and attempting to fill them.

The system currently relies on a feed to environmental posters, with this feed currently being a directory containing images of game posters. A future version could contain a connection to a camera or multiple cameras, which have live feeds to a specific environment. These visual feeds could then be processed by our visual module to determine the contents of the environment in real time. This live feed could potentially be in an inaccessible environment, which can only be perceived through CCTV cameras. Our AI player could monitor several feeds concurrently, or we could have multiple AI players monitoring a single camera, each reporting its observations to the central agent.

From an implementation standpoint, the visual module would need a more efficient computer vision solution to perform image recognition in real time. For example the ORB, FAST and SURF descriptors could be investigated as possible solutions. This is due to their superior speed and similar performance to the SIFT descriptor.

Further research into computer vision could allow me to experiment with classifiers that can detect on the HSV (Hue, Saturation, Values) spectrum in images. The current implementation focuses on the intensity of the image in grayscale, disregarding RGB values. A HSV classifier has the ability to detect colour space by separating luma (image intensity) from chroma (colourfulness). Using a HSV detector may be beneficial when performing real time image detection in dynamic environments. This is due to its robustness to lighting changes relating to shadows, and differing levels of brightness during the day.

A final area that could be extended on is the extraction of knowledge that does not have an explicit question relating to it. Currently only questions within the game can be answered, meaning the extra objects (Gorilla, Robot, Balloon, Ghost) cannot be reported on. These objects have been trained to be recognised with the SIFT descriptor however, with their parameters present in the model.

7.0 Reflection

In this section I'll provide a reflection on the development of the AI player, including implementation, development processes and technologies used.

I utilised an agile approach to developing this project, and on reflection this was the correct decision. The agile approach allowed me to efficiently deal with specification and functionality changes which were expected following supervisor meetings. For example, an early implementation of the AI player focused on taking images of scenes with a smartphone camera. Acting more as a cognitive assistant at the time, a prototype of this system was made and demonstrated. Following a supervisor meeting, the decision was taken to adapt the system into a full player, taking questions as input and receiving "feeds" to the scenes. The agile approach allowed me to succeed when responding to specification changes.

In hindsight, I believe CENode and its components were introduced far too late into the project lifecycle. This resulted in me having to re-implement some of the same functionality, but in a slightly different way. For example, the original training data was saved to a local JSON file and read into the system. Doing further research into CENode, I discovered that querying the CE models concepts and instances returns JSON encoded packets. I concluded it made sense to try and integrate my training data into the model itself, thus providing a unified location for retrieving external data. I believe this was the correct decision, as it provided a level of modularity between the system itself and its data sources.

The main disadvantage of receiving all model data in JSON however, is its flat structure made it difficult to find the correct resources for some types of question. Questions asking about location required extensive traversal of the model, and required some inelegant code to arrive at the correct answer. This code can be specifically found in the `process_question_data` method in the linguistics module. Implementing this method again, I would decouple functionality and explore more efficient ways of coming to the same answer.

I believe using Python was the correct decision to implement the AI player, however due to the system's dependency on the OpenCV package, portability is proving to be difficult. OpenCV is an extensive package and can take a significant amount of time to set up. For the AI player to be hosted on a system that's not my development machine, time will need to be taken to install all OpenCV dependencies. In the future, knowing the system is using the SIFT algorithm, I could try and bypass the OpenCV package, and implement a stand alone version of SIFT. This may prove difficult as there are no guarantees it'll interface easily with Python. In the meantime. the AI player can be launched externally by connecting via ssh into my development machine.

Reflecting on the visual module, I noticed a logical error while determining the best match of an object within a scene. Although it produced the correct result in this implementation, it is something I would change in a future version. The error relates to the processing of 20 best matches to determine a positive match. All objects maximum distance parameter were determined by the top 20 closest matches, being summed and then divided by 20. The correct classification for the lemon object however only produced 15 matches. This meant all 15 matches of the lemon object were divided by 20, producing an invalid average. In a future version I'll remove the dependency on 20 matches, and instead implement a more dynamic approach. This dynamic approach would use percentages (eg: 80% of best matches) so a valid average is always produced.

8.0 Conclusion

To conclude, it is possible for an AI player to play the SHERLOCK tactical intelligence game alongside human players. It successfully achieves this by perceiving its environment, reasoning, describing and articulating what it observes in CNL strings. It can then communicate its observation to the central agent, which'll be relayed to other players.

In the most part the system components have been successfully modelled off the ACT-R cognitive architecture, implementing a linguistic module, visual module and communication module. Although we are missing a declarative memory, the system still possess a short term memory which it'll use to communicate an observation before forgetting it. The system can also perceive its immediate visual environment, and receive direct question input, however it cannot currently receive other players' environmental observations.

The implementation of the AI player has left much room for future work. The most immediate area of future work I would like to explore is having the AI player, play alongside real humans in a controlled SHERLOCK experiment in the Queens Building. Having it operational in a controlled environment, I'll be able to monitor its effectiveness when playing alongside human players. To make it more realistic, we could limit the scenes the AI player has access to, limiting its scope to only answer a subset of game questions. If we envision the AI player only has access to a certain amount of visual information, then in a real life setting it can only answer questions within its visual scope.

Overall I believe this project has proved there is scope to implement virtual entities when crowdsourcing knowledge. Although there is much work to be done in monitoring its effectiveness, this project proves it's possible.

9.0 References

[1] A Preece, W Webberley, D Braines, N Hu, T La Porta, E Zaroukian,and J Z Bakdash, SHERLOCK: Simple Human Experiments Regarding Locally Observed Collective Knowledge, December 2015.

[2] "ACT-R", *Act-r.psy.cmu.edu*, 2017. [Online]. Available: http://act-r.psy.cmu.edu/about/. [Accessed: 07- Apr- 2017].

[3] P. Langley, J. Laird and S. Rogers, "Cognitive architectures: Research issues and challenges", *Cognitive Systems Research*, vol. 10, no. 2, pp. 141-160, 2009.

[4] P. Rosenbloom, "Cognitive/Virtual Human Architecture", *http://ict.usc.edu*, 2017. [Online]. Available: http://ict.usc.edu/wp-content/uploads/overviews/CognitiveVirtual%20Human%20Architecture_Overview.pdf. [Accessed: 07- Apr- 2017].

[5] "Soar Home - Soar Cognitive Architecture", *Soar.eecs.umich.edu*, 2017. [Online]. Available: http://soar.eecs.umich.edu/. [Accessed: 08- Apr- 2017].

[6] "The CLARION Cognitive Architecture Project", Sites.google.com, 2017. [Online]. Available: https://sites.google.com/site/clarioncognitivearchitecture/. [Accessed: 08- Apr- 2017].

[7] “ACT-R diagram.”, *http://act-r.psy.cmu.edu/wordpress/wp-content/uploads/2012/09/buffers.gif*. 2017.

[8] "What is computer vision?", Bmva.org, 2017. [Online]. Available: http://www.bmva.org/visionoverview. [Accessed: 09- Apr- 2017].

[9] U. Sinha, "SIFT: Theory and Practice: Introduction - AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://www.aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/. [Accessed: 12- Apr- 2017].

[10] U. Sinha, "SIFT: Theory and Practice: The scale space - AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://www.aishack.in/tutorials/sift-scale-invariant-feature-transform-scale-space/. [Accessed: 12- Apr- 2017].

[11] U. Sinha, "SIFT: Theory and Practice: LoG approximations - AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://www.aishack.in/tutorials/sift-scale-invariant-feature-transform-log-approximation/. [Accessed: 12- Apr- 2017].

[12] "Taylor Series -- from Wolfram MathWorld", *Mathworld.wolfram.com*, 2017. [Online]. Available: http://mathworld.wolfram.com/TaylorSeries.html. [Accessed: 12- Apr- 2017].

[13] U. Sinha, "SIFT: Theory and Practice: Finding keypoints - AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://www.aishack.in/tutorials/sift-scale-invariant-feature-transform-keypoints/. [Accessed: 12- Apr- 2017].

[14] U. Sinha, "SIFT: Theory and Practice: Getting rid of low contrast keypoints - AI Shack -

Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://www.aishack.in/tutorials/sift-scale-invariant-feature-transform-eliminate-low-contrast/. [Accessed: 12- Apr- 2017].

[15] U. Sinha, "SIFT: Theory and Practice: Keypoint orientations - AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://www.aishack.in/tutorials/sift-scale-invariant-feature-transform-keypoint-orientation/. [Accessed: 12- Apr- 2017].

[16] U. Sinha, "SIFT: Theory and Practice: Generating a feature - AI Shack - Tutorials for OpenCV, computer vision, deep learning, image processing, neural networks and artificial intelligence.", *Aishack.in*, 2017. [Online]. Available: http://aishack.in/tutorials/sift-scale-invariant-feature-transform-features/. [Accessed: 12- Apr- 2017].

[17] "OpenCV library", *Opencv.org*, 2017. [Online]. Available: http://opencv.org/. [Accessed: 12- Apr- 2017].

[18] "Patent US6711293 - Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image", *Google Books*, 2017. [Online]. Available: https://www.google.com/patents/US6711293. [Accessed: 12- Apr- 2017].

[19] D. Lowe, "The SIFT (Scale Invariant Feature Transform) Detector and Descriptor", University of British Columbia, 1999.

[20] L. Terveen. Overview of human-computer collaboration. Knowledge-Based Systems, 67–69, 1995.

[21] T. Kuhn, "A Survey and Classification of Controlled Natural Languages", *Computational Linguistics*, vol. 40, no. 1, pp. 121-170, 2014.

[22] "IBM Controlled Natural Language Processing Environment", *Ibm.com*, 2017. [Online]. Available: https://www.ibm.com/developerworks/community/groups/service/html/communitystart?communityUuid=558d55b6-78b6-43e6-9c14-0792481e4532. [Accessed: 13- Apr- 2017].

[23] T. Kuhn, A Survey and Classification of Controlled Natural Languages. [Online]. Available: http://www.aclweb.org/anthology/J14-1005. [Accessed: 13- Apr- 2017].

[24] "CE Store", *developerWorks Open*, 2017. [Online]. Available: https://developer.ibm.com/open/openprojects/ce-store/. [Accessed: 14- Apr- 2017].

[25] "CENode", *Cenode.io*, 2017. [Online]. Available: http://cenode.io/. [Accessed: 14- Apr- 2017]

[26] W. Webberley, A. Preece and D. Braines, "CENode: Enabling Human-Machine Conversations at the Network Edge", 2015.

[27] "pip 9.0.1 : Python Package Index", *Pypi.python.org*, 2017. [Online]. Available: https://pypi.python.org/pypi/pip. [Accessed: 20- Apr- 2017].

[28] Slack, "Slack: Where work happens," Slack. [Online]. Available: https://slack.com/. [Accessed: 20- Apr- 2017].

[29] "OpenCV: OpenCV-Python Tutorials", *Docs.opencv.org*, 2017. [Online]. Available: http://docs.opencv.org/3.2.0/d6/d00/tutorial_py_root.html. [Accessed: 21- Apr- 2017].

[30] "Requests: HTTP for Humans — Requests 2.13.0 documentation", *Docs.python-requests.org*, 2017. [Online]. Available: http://docs.python-requests.org/en/master/. [Accessed: 21- Apr- 2017].

[31] "Introduction — telepot 10.5 documentation", *Telepot.readthedocs.io*, 2017. [Online]. Available: http://telepot.readthedocs.io/en/latest/. [Accessed: 21- Apr- 2017].

[32] A. Rosebrock, "Ubuntu 16.04: How to install OpenCV - PyImageSearch", PyImageSearch, 2017. [Online]. Available: http://www.pyimagesearch.com/2016/10/24/ubuntu-16-04-how-to-install-opencv/. [Accessed: 24- Apr- 2017].

[33] "FlannBasedMatcher Python Fix (Fixes #5667) by patricksnape · Pull Request #6009 · opencv/opencv", GitHub, 2017. [Online]. Available: https://github.com/opencv/opencv/pull/6009. [Accessed: 24- Apr- 2017].

[34] "Feature Matching — OpenCV-Python Tutorials 1 documentation", Opencv-python-tutroals.readthedocs.io, 2017. [Online]. Available: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html#flann-based-matcher. [Accessed: 26- Apr- 2017].

[35] D. Marius Muja, "Fast approximate nearest neighbors with automatic algorithm configuration", Citeseer.ist.psu.edu, 2017. [Online]. Available: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.1721. [Accessed: 26- Apr- 2017].

[36] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints", International Journal of Computer Vision, vol. 60, no. 2, pp. 91-110, 2004.

10.0 Appendices

Appendix 1 - Original SHERLOCK CE model

```
conceptualise a ~ sherlock thing ~ S that is an entity and is an imageable thing
conceptualise an ~ organisation ~ O that is a sherlock thing
conceptualise a ~ fruit ~ F that is a sherlock thing and is a locatable thing
conceptualise a ~ room ~ R that is a location and is a sherlock thing
conceptualise a ~ hat colour ~ C
conceptualise a ~ sport ~ S
conceptualise a ~ character ~ C that is a sherlock thing and is a locatable thing and has the hat colour C as ~ hat colour ~
conceptualise the character C ~ works for ~ the organisation O and ~ eats ~ the fruit F and ~ plays ~ the sport S
conceptualise the hat colour C ~ is worn by ~ the character C
conceptualise an ~ object ~ O that is an entity
conceptualise the object O ~ resides in ~ the room R
conceptualise the room R ~ contains ~ the fruit F and has the character C as ~ contents ~ and has the object O as ~ additional contents ~
conceptualise the fruit F ~ is eaten by ~ the character C
conceptualise the sport S ~ is played by ~ the character C and ~ is in ~ the room R
conceptualise a ~ question ~ Q that has the value V as ~ text ~ and has the value W as ~ value ~ and has the value X as ~ relationship ~
conceptualise the question Q ~ concerns ~ the sherlock thing C
there is a rule named r1 that has 'if the character C ~ eats ~ the fruit F then the fruit F ~ is eaten by ~ the character C' as instruction
there is a rule named r2 that has 'if the character C ~ plays ~ the sport S then the sport S ~ is played by ~ the character C' as instruction
there is a rule named r3 that has 'if the character C has the hat colour S as ~ hat colour ~ then the hat colour S ~ is worn by ~ the character C' as instruction
there is a rule named r4 that has 'if the character C ~ is in ~ the room R then the room R has the character C as ~ contents ~' as instruction
there is a rule named r5 that has 'if the fruit F ~ is in ~ the room R then the room R ~ contains ~ the fruit F' as instruction
there is a rule named r6 that has 'if the fruit F ~ is eaten by ~ the character C then the character C ~ eats ~ the fruit F' as instruction
there is a rule named r7 that has 'if the sport S ~ is played by ~ the character C then the character C ~ plays ~ the sport S' as instruction
there is a rule named r8 that has 'if the hat colour S ~ is worn by ~ the character C then the character C has the hat colour S as ~ hat colour ~' as instruction
there is a rule named r9 that has 'if the room R has the character C as ~ contents ~ then the character C ~ is in ~ the room R' as instruction
there is a rule named r10 that has 'if the room R ~ contains ~ the fruit F then the fruit F ~ is in ~ the room R' as instruction
there is a character named 'Prof Crane' that has 'http://sherlock.cenode.io/media/crane.png' as image
there is a character named 'Dr Finch' that has 'http://sherlock.cenode.io/media/finch.png' as image
there is a character named 'Col Robin' that has 'http://sherlock.cenode.io/media/robin.png' as image
there is a character named 'Sgt Stork' that has 'http://sherlock.cenode.io/media/stork.png' as image
there is a character named 'Rev Hawk' that has 'http://sherlock.cenode.io/media/hawk.png' as image
```

```
  there is a character named 'Capt Falcon' that has 'http://sherlock.cenode.io/media/falcon.png' as image
  there is a character named 'Elephant' that has 'http://sherlock.cenode.io/media/Elephant.png' as image
  there is a character named 'Giraffe' that has 'http://sherlock.cenode.io/media/Giraffe.png' as image
  there is a character named 'Hippopotamus' that has 'http://sherlock.cenode.io/media/Hippopotamus.png' as image
  there is a character named 'Leopard' that has 'http://sherlock.cenode.io/media/Leopard.png' as image
  there is a character named 'Lion' that has 'http://sherlock.cenode.io/media/Lion.png' as image
  there is a character named 'Zebra' that has 'http://sherlock.cenode.io/media/Zebra.png' as image
  there is a room named 'Ruby Room'
  there is a room named 'Sapphire Room'
  there is a room named 'Gold Room'
  there is a room named 'Amber Room'
  there is a room named 'Emerald Room'
  there is a room named 'Silver Room'
  there is a fruit named 'pineapple'
  there is a fruit named 'apple'
  there is a fruit named 'banana'
  there is a fruit named 'orange'
  there is a fruit named 'lemon'
  there is a fruit named 'pear'
  there is a fruit named 'grape'
  there is a fruit named 'kiwi'
  there is a fruit named 'tomato'
  there is a hat colour named 'green'
  there is a hat colour named 'red'
  there is a hat colour named 'yellow'
  there is a hat colour named 'black'
  there is a hat colour named 'white'
  there is a hat colour named 'purple'
  there is a hat colour named 'pink'
  there is a hat colour named 'blue'
  there is a hat colour named 'brown'
  there is a hat colour named 'grey'
  there is a sport named 'tennis'
  there is a sport named 'badminton'
  there is a sport named 'rugby'
  there is a sport named 'football'
  there is a sport named 'soccer'
  there is a sport named 'running'
  there is a sport named 'swimming'
  there is a sport named 'athletics'
  there is a sport named 'baseball'
  there is a sport named 'rounders'
  there is a sport named 'softball'
  there is a sport named 'cricket'
  there is a sport named 'golf'
  there is a sport named 'basketball'
  there is a rule named objectrule1 that has 'if the object O ~ resides in ~ the room R then the room R has the object O as ~ additional contents ~' as instruction
  there is an object named 'gorilla'
  there is an object named 'dinosaur'
  there is an object named 'robot'
  there is an object named 'ghost'
  there is an object named 'balloon'
  there is a question named 'q1' that has 'What character eats pineapples?' as text and has 'is eaten by' as relationship and concerns the fruit 'pineapple'
  there is a question named 'q2' that has 'What sport does Zebra play?' as text and has 'plays' as relationship and concerns the character 'Zebra'
  there is a question named 'q3' that has 'What character eats apples?' as text and has 'is eaten by' as relationship and concerns the fruit 'apple'
  there is a question named 'q4' that has 'What colour hat is Elephant wearing?' as text and has 'hat colour' as value and concerns the character 'Elephant'
  there is a question named 'q6' that has 'Where is Giraffe?' as text and has 'is in' as relationship and concerns the character 'Giraffe'
  there is a question named 'q7' that has 'What colour hat is Lion wearing?' as text and has 'hat colour' as value and concerns the character 'Lion'
  there is a question named 'q8' that has 'Where is Lion?' as text and has 'is in' as relationship and concerns the character 'Lion'
  there is a question named 'q9' that has 'Which character is in the emerald room?' as text and has 'contents' as value and concerns the room 'Emerald Room'
  there is a question named 'q12' that has 'What character eats bananas?' as text and has 'is eaten by' as relationship and concerns the fruit 'banana'
  there is a question named 'q13' that has 'What character is in the sapphire room?' as text and has 'contents' as value and concerns the room 'Sapphire Room'
  there is a question named 'q17' that has 'What sport does Elephant play?' as text and has 'plays' as relationship and concerns the character 'Elephant'
  there is a question named 'q18' that has 'What character is wearing a red hat?' as text and has 'is worn by' as relationship and concerns the hat colour 'red'
  there is a question named 'q19' that has 'What character plays rugby?' as text and has 'is played by' as relationship and concerns the sport 'rugby'
  there is a question named 'q20' that has 'What fruit does Leopard eat?' as text and has 'eats' as relationship and concerns the character 'Leopard'
  there is a question named 'q23' that has 'What fruit does Giraffe eat?' as text and has 'eats' as relationship and concerns the character 'Giraffe'
  there is a question named 'q24' that has 'What colour hat is Zebra wearing?' as text and has 'hat colour' as value and concerns the character 'Zebra'
  there is a question named 'q25' that has 'Where is the apple?' as text and has 'is in' as relationship and concerns the fruit 'apple'
  there is a question named 'q26' that has 'What character is wearing a yellow hat?' as text and has 'is worn by' as relationship and concerns the hat colour 'yellow'
  there is a question named 'q28' that has 'What fruit is in the silver room?' as text and has 'contains' as relationship and concerns the room 'Silver Room'
  there is a question named 'q30' that has 'What character is wearing a blue hat?' as text and has 'is worn by' as relationship and concerns the hat colour 'blue'
  there is a question named 'q31' that has 'What character eats lemons?' as text and has 'is eaten by' as relationship and concerns the fruit 'lemon'
  there is a question named 'q33' that has 'What fruit does Elephant eat?' as text and has 'eats' as relationship and concerns the character 'Elephant'
  there is a question named 'q34' that has 'What character plays basketball?' as text and has 'is played by' as relationship and concerns the sport 'basketball'
  there is a question named 'q35' that has 'What character plays soccer?' as text and has 'is played by' as relationship and concerns the sport 'soccer'
  there is a question named 'q36' that has 'What sport does Lion play?' as text and has 'plays' as relationship and concerns the character 'Lion'
  there is a question named 'q37' that has 'What character is in the ruby room?' as text and has 'contents' as value and concerns the room 'Ruby Room'
  there is a question named 'q39' that has 'What character plays golf?' as text and has 'is played by' as relationship and concerns the sport 'golf'
  there is a question named 'q40' that has 'What character eats oranges?' as text and has 'is eaten by' as relationship and concerns the fruit 'orange'
  there is a question named 'q41' that has 'What colour hat is Hippopotamus wearing?' as text and has 'hat colour' as value and concerns the character
'Hippopotamus'
  there is a question named 'q45' that has 'What character is in the amber room?' as text and has 'contents' as value and concerns the room 'Amber Room'
  there is a question named 'q47' that has 'Where is Elephant?' as text and has 'is in' as relationship and concerns the character 'Elephant'
  there is a question named 'q48' that has 'Where is the pear?' as text and has 'is in' as relationship and concerns the fruit 'pear'
  there is a question named 'q50' that has 'What fruit does Lion eat?' as text and has 'eats' as relationship and concerns the character 'Lion'
  there is a question named 'q52' that has 'What sport does Giraffe play?' as text and has 'plays' as relationship and concerns the character 'Giraffe'
  there is a question named 'q53' that has 'Where is Hippopotamus?' as text and has 'is in' as relationship and concerns the character 'Hippopotamus'
  there is a question named 'q54' that has 'What sport does Hippopotamus play?' as text and has 'plays' as relationship and concerns the character 'Hippopotamus'
```

Appendix 2 - Modified SHERLOCK CE model

```
  conceptualise a ~ sherlock thing ~ S that is an entity and is an imageable thing
  conceptualise a ~ fruit ~ F that is a sherlock thing and is a locatable thing
  conceptualise a ~ room ~ R that is a location and is a sherlock thing
  conceptualise a ~ hat colour ~ C that is a sherlock thing
  conceptualise a ~ sport ~ S that is a sherlock thing
  conceptualise a ~ character ~ C that is a sherlock thing and is a locatable thing and has the hat colour C as ~ hat colour ~
  conceptualise the character C ~ eats ~ the fruit F and ~ plays ~ the sport S
  conceptualise the hat colour C ~ is worn by ~ the character C
  conceptualise an ~ object ~ O that is an entity and is a sherlock thing
  conceptualise the object O ~ resides in ~ the room R
  conceptualise the room R ~ contains ~ the fruit F and has the character C as ~ contents ~ and has the object O as ~ additional contents ~
  conceptualise the fruit F ~ is eaten by ~ the character C
  conceptualise the sport S ~ is played by ~ the character C and ~ is in ~ the room R
  conceptualise a ~ question ~ Q that has the value V as ~ text ~ and has the value W as ~ value ~ and has the value X as ~ relationship ~
  conceptualise the question Q ~ concerns ~ the sherlock thing C
  conceptualise a ~ sift parameter set ~ P that has the sherlock thing T as ~ related thing ~ and has the value K as ~ maximum distance ~ and has the value J as ~
minimum matches ~
  there is a rule named r1 that has 'if the character C ~ eats ~ the fruit F then the fruit F ~ is eaten by ~ the character C' as instruction
  there is a rule named r2 that has 'if the character C ~ plays ~ the sport S then the sport S ~ is played by ~ the character C' as instruction
  there is a rule named r3 that has 'if the character C has the hat colour S as ~ hat colour ~ then the hat colour S ~ is worn by ~ the character C' as instruction
  there is a rule named r4 that has 'if the character C ~ is in ~ the room R then the room R has the character C as ~ contents ~' as instruction
  there is a rule named r5 that has 'if the fruit F ~ is in ~ the room R then the room R ~ contains ~ the fruit F' as instruction
  there is a rule named r6 that has 'if the fruit F ~ is eaten by ~ the character C then the character C ~ eats ~ the fruit F' as instruction
  there is a rule named r7 that has 'if the sport S ~ is played by ~ the character C then the character C ~ plays ~ the sport S' as instruction
  there is a rule named r8 that has 'if the hat colour S ~ is worn by ~ the character C then the character C has the hat colour S as ~ hat colour ~' as instruction
  there is a rule named r9 that has 'if the room R has the character C as ~ contents ~ then the character C ~ is in ~ the room R' as instruction
  there is a rule named r10 that has 'if the room R ~ contains ~ the fruit F then the fruit F ~ is in ~ the room R' as instruction
  there is a character named 'Elephant' that has 'elephant.jpg' as image
  there is a character named 'Giraffe' that has 'giraffe.jpg' as image
  there is a character named 'Hippopotamus' that has 'hippo.jpg' as image
  there is a character named 'Leopard' that has 'leopard.jpg' as image
  there is a character named 'Lion' that has 'lion.jpg' as image
  there is a character named 'Zebra' that has 'zebra.jpg' as image
  there is a room named 'Ruby Room' that has 'ruby_room.jpg' as image
  there is a room named 'Sapphire Room' that has 'saphire_room.jpg' as image
  there is a room named 'Gold Room' that has 'gold_room.jpg' as image
  there is a room named 'Amber Room' that has 'amber_room.jpg' as image
  there is a room named 'Emerald Room' that has 'emerald_room.jpg' as image
  there is a room named 'Silver Room' that has 'silver_room.jpg' as image
  there is a fruit named 'pineapple' that has 'pineapple.jpg' as image
  there is a fruit named 'apple' that has 'apple.jpg' as image
  there is a fruit named 'banana' that has 'bananas.jpg' as image
  there is a fruit named 'orange' that has 'orange.jpg' as image
  there is a fruit named 'lemon' that has 'lemon.jpg' as image
  there is a fruit named 'pear' that has 'pear.JPG' as image
  there is a hat colour named 'green' that has 'green_hat.jpg' as image
  there is a hat colour named 'red' that has 'red_hat.jpg' as image
  there is a hat colour named 'yellow' that has 'yellow_hat.jpg' as image
  there is a hat colour named 'purple' that has 'purple_hat.jpg' as image
  there is a hat colour named 'pink' that has 'pink_hat.jpg' as image
  there is a hat colour named 'blue' that has 'blue_hat.jpg' as image
  there is a sport named 'tennis' that has 'tennis.jpg' as image
  there is a sport named 'rugby' that has 'rugby.jpg' as image
  there is a sport named 'soccer' that has 'football.jpg' as image
  there is a sport named 'cricket' that has 'cricket.jpg' as image
  there is a sport named 'golf' that has 'golf.jpg' as image
  there is a sport named 'basketball' that has 'basketball.jpg' as image
  there is a rule named objectrule1 that has 'if the object O ~ resides in ~ the room R then the room R has the object O as ~ additional contents ~' as instruction
  there is an object named 'gorilla' that has 'gorilla.jpg' as image
  there is an object named 'robot' that has 'robot.jpg' as image
  there is an object named 'ghost' that has 'ghost.jpg' as image
  there is an object named 'balloon' that has 'balloon.jpg' as image
  there is a question named 'q1' that has 'What character eats pineapples?' as text and has 'is eaten by' as relationship and concerns the fruit 'pineapple'
  there is a question named 'q2' that has 'What sport does Zebra play?' as text and has 'plays' as relationship and concerns the character 'Zebra'
  there is a question named 'q3' that has 'What character eats apples?' as text and has 'is eaten by' as relationship and concerns the fruit 'apple'
  there is a question named 'q4' that has 'What colour hat is Elephant wearing?' as text and has 'hat colour' as value and concerns the character 'Elephant'
  there is a question named 'q6' that has 'Where is Giraffe?' as text and has 'is in' as relationship and concerns the character 'Giraffe'
  there is a question named 'q7' that has 'What colour hat is Lion wearing?' as text and has 'hat colour' as value and concerns the character 'Lion'
  there is a question named 'q8' that has 'Where is Lion?' as text and has 'is in' as relationship and concerns the character 'Lion'
  there is a question named 'q9' that has 'Which character is in the emerald room?' as text and has 'contents' as value and concerns the room 'Emerald Room'
  there is a question named 'q12' that has 'What character eats bananas?' as text and has 'is eaten by' as relationship and concerns the fruit 'banana'
  there is a question named 'q13' that has 'What character is in the sapphire room?' as text and has 'contents' as value and concerns the room 'Sapphire Room'
  there is a question named 'q17' that has 'What sport does Elephant play?' as text and has 'plays' as relationship and concerns the character 'Elephant'
  there is a question named 'q18' that has 'What character is wearing a red hat?' as text and has 'is worn by' as relationship and concerns the hat colour 'red'
  there is a question named 'q19' that has 'What character plays rugby?' as text and has 'is played by' as relationship and concerns the sport 'rugby'
  there is a question named 'q20' that has 'What fruit does Leopard eat?' as text and has 'eats' as relationship and concerns the character 'Leopard'
  there is a question named 'q23' that has 'What fruit does Giraffe eat?' as text and has 'eats' as relationship and concerns the character 'Giraffe'
  there is a question named 'q24' that has 'What colour hat is Zebra wearing?' as text and has 'hat colour' as value and concerns the character 'Zebra'
  there is a question named 'q25' that has 'Where is the apple?' as text and has 'is in' as relationship and concerns the fruit 'apple'
  there is a question named 'q26' that has 'What character is wearing a yellow hat?' as text and has 'is worn by' as relationship and concerns the hat colour 'yellow'
  there is a question named 'q28' that has 'What fruit is in the silver room?' as text and has 'contains' as relationship and concerns the room 'Silver Room'
  there is a question named 'q30' that has 'What character is wearing a blue hat?' as text and has 'is worn by' as relationship and concerns the hat colour 'blue'
  there is a question named 'q31' that has 'What character eats lemons?' as text and has 'is eaten by' as relationship and concerns the fruit 'lemon'
  there is a question named 'q33' that has 'What fruit does Elephant eat?' as text and has 'eats' as relationship and concerns the character 'Elephant'
  there is a question named 'q34' that has 'What character plays basketball?' as text and has 'is played by' as relationship and concerns the sport 'basketball'
  there is a question named 'q35' that has 'What character plays soccer?' as text and has 'is played by' as relationship and concerns the sport 'soccer'
  there is a question named 'q36' that has 'What sport does Lion play?' as text and has 'plays' as relationship and concerns the character 'Lion'
  there is a question named 'q37' that has 'What character is in the ruby room?' as text and has 'contents' as value and concerns the room 'Ruby Room'
  there is a question named 'q39' that has 'What character plays golf?' as text and has 'is played by' as relationship and concerns the sport 'golf'
  there is a question named 'q40' that has 'What character eats oranges?' as text and has 'is eaten by' as relationship and concerns the fruit 'orange'
  there is a question named 'q41' that has 'What colour hat is Hippopotamus wearing?' as text and has 'hat colour' as value and concerns the character
'Hippopotamus'
  there is a question named 'q45' that has 'What character is in the amber room?' as text and has 'contents' as value and concerns the room 'Amber Room'
  there is a question named 'q47' that has 'Where is Elephant?' as text and has 'is in' as relationship and concerns the character 'Elephant'
  there is a question named 'q48' that has 'Where is the pear?' as text and has 'is in' as relationship and concerns the fruit 'pear'
  there is a question named 'q50' that has 'What fruit does Lion eat?' as text and has 'eats' as relationship and concerns the character 'Lion'
  there is a question named 'q52' that has 'What sport does Giraffe play?' as text and has 'plays' as relationship and concerns the character 'Giraffe'
  there is a question named 'q53' that has 'Where is Hippopotamus?' as text and has 'is in' as relationship and concerns the character 'Hippopotamus'
  there is a question named 'q54' that has 'What sport does Hippopotamus play?' as text and has 'plays' as relationship and concerns the character 'Hippopotamus'
  there is a sift parameter set named 'Elephant parameters' that has the character 'Elephant' as related thing and has '45' as maximum distance and has '65' as
minimum matches
  there is a sift parameter set named 'Giraffe parameters' that has the character 'Giraffe' as related thing and has '40' as maximum distance and has '130' as minimum
matches
  there is a sift parameter set named 'Hippopotamus parameters' that has the character 'Hippopotamus' as related thing and has '35' as maximum distance and has
'105' as minimum matches
  there is a sift parameter set named 'Leopard parameters' that has the character 'Leopard' as related thing and has '55' as maximum distance and has '200' as
minimum matches
  there is a sift parameter set named 'Lion parameters' that has the character 'Lion' as related thing and has '55' as maximum distance and has '95' as minimum
matches
  there is a sift parameter set named 'Zebra parameters' that has the character 'Zebra' as related thing and has '35' as maximum distance and has '185' as minimum
matches
  there is a sift parameter set named 'amber room parameters' that has the room 'amber room' as related thing and has '32' as maximum distance and has '175' as
minimum matches
  there is a sift parameter set named 'emerald room parameters' that has the room 'emerald room' as related thing and has '35' as maximum distance and has '190' as
minimum matches
  there is a sift parameter set named 'gold room parameters' that has the room 'gold room' as related thing and has '30' as maximum distance and has '155' as
minimum matches
  there is a sift parameter set named 'ruby room parameters' that has the room 'ruby room' as related thing and has '10' as maximum distance and has '155' as
minimum matches
  there is a sift parameter set named 'sapphire room parameters' that has the room 'sapphire room' as related thing and has '5' as maximum distance and has '200' as
minimum matches
  there is a sift parameter set named 'silver room parameters' that has the room 'silver room' as related thing and has '5' as maximum distance and has '190' as
minimum matches
  there is a sift parameter set named 'apple parameters' that has the fruit 'apple' as related thing and has '90' as maximum distance and has '30' as minimum matches
  there is a sift parameter set named 'banana parameters' that has the fruit 'banana' as related thing and has '40' as maximum distance and has '70' as minimum
matches
  there is a sift parameter set named 'lemon parameters' that has the fruit 'lemon' as related thing and has '26' as maximum distance and has '12' as minimum
matches
  there is a sift parameter set named 'orange parameters' that has the fruit 'orange' as related thing and has '55' as maximum distance and has '30' as minimum
matches
  there is a sift parameter set named 'pineapple parameters' that has the fruit 'pineapple' as related thing and has '20' as maximum distance and has '45' as minimum
matches
  there is a sift parameter set named 'pear parameters' that has the fruit 'pear' as related thing and has '75' as maximum distance and has '55' as minimum matches
  there is a sift parameter set named 'basketball parameters' that has the sport 'basketball' as related thing and has '15' as maximum distance and has '35' as
minimum matches
  there is a sift parameter set named 'cricket parameters' that has the sport 'cricket' as related thing and has '45' as maximum distance and has '55' as minimum
matches
  there is a sift parameter set named 'soccer parameters' that has the sport 'soccer' as related thing and has '65' as maximum distance and has '40' as minimum
matches
  there is a sift parameter set named 'golf parameters' that has the sport 'golf' as related thing and has '35' as maximum distance and has '50' as minimum matches
  there is a sift parameter set named 'rugby parameters' that has the sport 'rugby' as related thing and has '30' as maximum distance and has '125' as minimum
matches
  there is a sift parameter set named 'tennis parameters' that has the sport 'tennis' as related thing and has '30' as maximum distance and has '110' as minimum
matches
  there is a sift parameter set named 'red parameters' that has the hat colour 'red' as related thing and has '55' as maximum distance and has '50' as minimum
matches
  there is a sift parameter set named 'green parameters' that has the colour 'green' as related thing and has '85' as maximum distance and has '25' as minimum
matches
  there is a sift parameter set named 'yellow parameters' that has the colour 'yellow' as related thing and has '20' as maximum distance and has '40' as minimum
matches
  there is a sift parameter set named 'blue parameters' that has the colour 'blue' as related thing and has '25' as maximum distance and has '55' as minimum matches
  there is a sift parameter set named 'pink parameters' that has the colour 'pink' as related thing and has '40' as maximum distance and has '50' as minimum matches
  there is a sift parameter set named 'purple parameters' that has the colour 'purple' as related thing and has '65' as maximum distance and has '45' as minimum
matches
  there is a sift parameter set named 'balloon parameters' that has the object 'balloon' as related thing and has '65' as maximum distance and has '120' as minimum
matches
  there is a sift parameter set named 'ghost parameters' that has the object 'ghost' as related thing and has '220' as maximum distance and has '18' as minimum
matches
  there is a sift parameter set named 'gorilla parameters' that has the object 'gorilla' as related thing and has '30' as maximum distance and has '115' as minimum
matches
  there is a sift parameter set named 'robot parameters' that has the object 'robot' as related thing and has '20' as maximum distance and has '220' as minimum
matches
```

Appendix 3 - Query of instances with SHERLOCK CE model uploaded

```
[{"name":"Moira","id":1,"conceptName":"agent","conceptId":4},{"name":"r1","id":2,"conceptName":"rule","conceptId":14},{"name":"r2","i
d":3,"conceptName":"rule","conceptId":14},{"name":"r3","id":4,"conceptName":"rule","conceptId":14},{"name":"r4","id":5,"conceptName":
"rule","conceptId":14},{"name":"r5","id":6,"conceptName":"rule","conceptId":14},{"name":"r6","id":7,"conceptName":"rule","conceptId":
14},{"name":"r7","id":8,"conceptName":"rule","conceptId":14},{"name":"r8","id":9,"conceptName":"rule","conceptId":14},{"name":"r9","i
d":10,"conceptName":"rule","conceptId":14},{"name":"r10","id":11,"conceptName":"rule","conceptId":14},{"name":"Elephant","id":12,"con
ceptName":"character","conceptId":27},{"name":"Giraffe","id":13,"conceptName":"character","conceptId":27},{"name":"Hippopotamus","id"
:14,"conceptName":"character","conceptId":27},{"name":"Leopard","id":15,"conceptName":"character","conceptId":27},{"name":"Lion","id"
:16,"conceptName":"character","conceptId":27},{"name":"Zebra","id":17,"conceptName":"character","conceptId":27},{"name":"Ruby
Room","id":18,"conceptName":"room","conceptId":24},{"name":"Sapphire Room","id":19,"conceptName":"room","conceptId":24},{"name":"Gold
Room","id":20,"conceptName":"room","conceptId":24},{"name":"Amber Room","id":21,"conceptName":"room","conceptId":24},{"name":"Emerald
Room","id":22,"conceptName":"room","conceptId":24},{"name":"Silver
Room","id":23,"conceptName":"room","conceptId":24},{"name":"pineapple","id":24,"conceptName":"fruit","conceptId":23},{"name":"apple",
"id":25,"conceptName":"fruit","conceptId":23},{"name":"banana","id":26,"conceptName":"fruit","conceptId":23},{"name":"orange","id":27
,"conceptName":"fruit","conceptId":23},{"name":"lemon","id":28,"conceptName":"fruit","conceptId":23},{"name":"pear","id":29,"conceptN
ame":"fruit","conceptId":23},{"name":"green","id":30,"conceptName":"hat
colour","conceptId":25},{"name":"red","id":31,"conceptName":"hat colour","conceptId":25},{"name":"yellow","id":32,"conceptName":"hat
colour","conceptId":25},{"name":"purple","id":33,"conceptName":"hat colour","conceptId":25},{"name":"pink","id":34,"conceptName":"hat
colour","conceptId":25},{"name":"blue","id":35,"conceptName":"hat
colour","conceptId":25},{"name":"tennis","id":36,"conceptName":"sport","conceptId":26},{"name":"rugby","id":37,"conceptName":"sport",
"conceptId":26},{"name":"soccer","id":38,"conceptName":"sport","conceptId":26},{"name":"cricket","id":39,"conceptName":"sport","conce
ptId":26},{"name":"golf","id":40,"conceptName":"sport","conceptId":26},{"name":"basketball","id":41,"conceptName":"sport","conceptId"
:26},{"name":"objectrule1","id":42,"conceptName":"rule","conceptId":14},{"name":"gorilla","id":43,"conceptName":"object","conceptId":
28},{"name":"robot","id":44,"conceptName":"object","conceptId":28},{"name":"ghost","id":45,"conceptName":"object","conceptId":28},{"n
ame":"balloon","id":46,"conceptName":"object","conceptId":28},{"name":"q1","id":47,"conceptName":"question","conceptId":29},{"name":"
q2","id":48,"conceptName":"question","conceptId":29},{"name":"q3","id":49,"conceptName":"question","conceptId":29},{"name":"q4","id":
50,"conceptName":"question","conceptId":29},{"name":"q6","id":51,"conceptName":"question","conceptId":29},{"name":"q7","id":52,"conce
ptName":"question","conceptId":29},{"name":"q8","id":53,"conceptName":"question","conceptId":29},{"name":"q9","id":54,"conceptName":"
question","conceptId":29},{"name":"q12","id":55,"conceptName":"question","conceptId":29},{"name":"q13","id":56,"conceptName":"questio
n","conceptId":29},{"name":"q17","id":57,"conceptName":"question","conceptId":29},{"name":"q18","id":58,"conceptName":"question","con
ceptId":29},{"name":"q19","id":59,"conceptName":"question","conceptId":29},{"name":"q20","id":60,"conceptName":"question","conceptId"
:29},{"name":"q23","id":61,"conceptName":"question","conceptId":29},{"name":"q24","id":62,"conceptName":"question","conceptId":29},{"
name":"q25","id":63,"conceptName":"question","conceptId":29},{"name":"q26","id":64,"conceptName":"question","conceptId":29},{"name":"
q28","id":65,"conceptName":"question","conceptId":29},{"name":"q30","id":66,"conceptName":"question","conceptId":29},{"name":"q31","i
d":67,"conceptName":"question","conceptId":29},{"name":"q33","id":68,"conceptName":"question","conceptId":29},{"name":"q34","id":69,"
conceptName":"question","conceptId":29},{"name":"q35","id":70,"conceptName":"question","conceptId":29},{"name":"q36","id":71,"concept
Name":"question","conceptId":29},{"name":"q37","id":72,"conceptName":"question","conceptId":29},{"name":"q39","id":73,"conceptName":"
question","conceptId":29},{"name":"q40","id":74,"conceptName":"question","conceptId":29},{"name":"q41","id":75,"conceptName":"questio
n","conceptId":29},{"name":"q45","id":76,"conceptName":"question","conceptId":29},{"name":"q47","id":77,"conceptName":"question","con
ceptId":29},{"name":"q48","id":78,"conceptName":"question","conceptId":29},{"name":"q50","id":79,"conceptName":"question","conceptId"
:29},{"name":"q52","id":80,"conceptName":"question","conceptId":29},{"name":"q53","id":81,"conceptName":"question","conceptId":29},{"
name":"q54","id":82,"conceptName":"question","conceptId":29},{"name":"Elephant parameters","id":83,"conceptName":"sift parameter
set","conceptId":30},{"name":"Giraffe parameters","id":84,"conceptName":"sift parameter set","conceptId":30},{"name":"Hippopotamus
parameters","id":85,"conceptName":"sift parameter set","conceptId":30},{"name":"Leopard parameters","id":86,"conceptName":"sift
parameter set","conceptId":30},{"name":"Lion parameters","id":87,"conceptName":"sift parameter set","conceptId":30},{"name":"Zebra
parameters","id":88,"conceptName":"sift parameter set","conceptId":30},{"name":"amber room parameters","id":89,"conceptName":"sift
parameter set","conceptId":30},{"name":"emerald room parameters","id":90,"conceptName":"sift parameter
set","conceptId":30},{"name":"gold room parameters","id":91,"conceptName":"sift parameter set","conceptId":30},{"name":"ruby room
parameters","id":92,"conceptName":"sift parameter set","conceptId":30},{"name":"sapphire room parameters","id":93,"conceptName":"sift
parameter set","conceptId":30},{"name":"silver room parameters","id":94,"conceptName":"sift parameter
set","conceptId":30},{"name":"apple parameters","id":95,"conceptName":"sift parameter set","conceptId":30},{"name":"banana
parameters","id":96,"conceptName":"sift parameter set","conceptId":30},{"name":"lemon parameters","id":97,"conceptName":"sift
parameter set","conceptId":30},{"name":"orange parameters","id":98,"conceptName":"sift parameter
set","conceptId":30},{"name":"pineapple parameters","id":99,"conceptName":"sift parameter set","conceptId":30},{"name":"pear
parameters","id":100,"conceptName":"sift parameter set","conceptId":30},{"name":"basketball parameters","id":101,"conceptName":"sift
parameter set","conceptId":30},{"name":"cricket parameters","id":102,"conceptName":"sift parameter
set","conceptId":30},{"name":"soccer parameters","id":103,"conceptName":"sift parameter set","conceptId":30},{"name":"golf
parameters","id":104,"conceptName":"sift parameter set","conceptId":30},{"name":"rugby parameters","id":105,"conceptName":"sift
parameter set","conceptId":30},{"name":"tennis parameters","id":106,"conceptName":"sift parameter set","conceptId":30},{"name":"red
parameters","id":107,"conceptName":"sift parameter set","conceptId":30},{"name":"green parameters","id":108,"conceptName":"sift
parameter set","conceptId":30},{"name":"yellow parameters","id":109,"conceptName":"sift parameter set","conceptId":30},{"name":"blue
parameters","id":110,"conceptName":"sift parameter set","conceptId":30},{"name":"pink parameters","id":111,"conceptName":"sift
parameter set","conceptId":30},{"name":"purple parameters","id":112,"conceptName":"sift parameter
set","conceptId":30},{"name":"balloon parameters","id":113,"conceptName":"sift parameter set","conceptId":30},{"name":"ghost
parameters","id":114,"conceptName":"sift parameter set","conceptId":30},{"name":"gorilla parameters","id":115,"conceptName":"sift
parameter set","conceptId":30},{"name":"robot parameters","id":116,"conceptName":"sift parameter set","conceptId":30}]
```

Appendix 4 - Process Question Data Method

```
def process_question_data(self, subjectType, subjectName, relationship, questionID):
    """ Searches the model for other useful data relating to the question.
        different question formats have a different method of retreiving data.
        These questions ids have been placed in groups in the constructor. """

    objectType = ""

    try:

        # 'What colour hat is Elephant wearing?' type question
        if questionID in self.question_group_1:
            objectType = relationship
            for concepts in self.concept_data:
                if concepts["name"] == relationship:
                        subjectTypeID = concepts["id"]
                        subjectTypeConcept =
self.sherlock_model.get_model_concept_with_id(subjectTypeID)
                        relationship = subjectTypeConcept["relationships"][0]["label"]


            # "Where is the apple?"" type question
            elif questionID in self.question_group_2:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        concept_id = concepts["id"]

                subjectTypeConcept =
self.sherlock_model.get_model_concept_with_id(concept_id)

                for concepts in subjectTypeConcept["parents"]:
                    if concepts["name"] == "locatable thing":
                        concept_id = concepts["id"]

                subjectTypeConcept =
self.sherlock_model.get_model_concept_with_id(concept_id)

                for objectTypeValue in subjectTypeConcept["relationships"]:
                    locatableID = objectTypeValue["targetId"]

                objectTypeConcept =
self.sherlock_model.get_model_concept_with_id(locatableID)

                for objectTypeValue in objectTypeConcept["children"]:
                    objectType = objectTypeValue["name"]


            # 'Which character is in the emerald room?' type question
            elif questionID in self.question_group_3:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        subjectTypeID = concepts["id"]

                subjectTypeConcept =
self.sherlock_model.get_model_concept_with_id(subjectTypeID)

                for conceptValues in subjectTypeConcept["values"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        objectTypeID = conceptValues["targetId"]
                        break

                for conceptValues in subjectTypeConcept["relationships"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        objectTypeID = conceptValues["targetId"]
                        break

                objectTypeConcept =
self.sherlock_model.get_model_concept_with_id(objectTypeID)

                for objectTypeValue in objectTypeConcept["parents"]:
                    if objectTypeValue["name"] == "locatable thing":
                        locatableID = objectTypeValue["id"]

                objectTypeConcept =
self.sherlock_model.get_model_concept_with_id(locatableID)

                for objectTypeValue in objectTypeConcept["relationships"]:
                    relationship = objectTypeValue["label"]
                    locatableID = objectTypeValue["targetId"]

                objectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(locatableID)

                for objectTypeValue in objectTypeConcept["children"]:
                    subjectType = objectTypeValue["name"]


            # All other questions
            else:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        subjectTypeID = concepts["id"]

                subjectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(subjectTypeID)

                for conceptValues in subjectTypeConcept["relationships"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        break

            self.bot.sendMessage(self.chat_id, "We're looking for {0} associated with 
'{1}'".format(objectType, subjectName))
            print "We're looking for '{0}' associated with '{1}'".format(objectType, 
subjectName)

        except:
            print "Could not gather all data from the model"
            tb = traceback.format_exc()
            print tb
            return


        self.search_environment(objectType, subjectName, subjectType, relationship, 
questionID)
```

Appendix 5 - Full Code Base

```
#!usr/bin/python
import telepot
import time
import sys
import numpy as np
import cv2
import os
from os import listdir
from os.path import isfile, join, dirname, realpath
import json
import requests
import traceback

class telegram_handle:

    def __init__(self):
        # bot API key
        self.bot = telepot.Bot('353119581:AAGUpjclZ2RjWW-L-OrHU4bTHltqB1SGYFc')
        self.bot.message_loop(self.handle)
        self.trusted = ["robbbh"]
    def handle(self, msg):
        self.userName = msg["from"]["username"]
        self.chat_id = msg['chat']['id']

        if self.userName not in self.trusted:
            self.bot.sendMessage(self.chat_id, "Unverified User...\nPlease use a
verified account.")
            return
        else:
            self.check_message_type(msg)


    def check_message_type(self, msg):

        if "text" in msg:
            self.message = msg['text'].encode('utf-8').strip()
            recieved_text = "Recieved text input = '{0}'".format(self.message)
            print recieved_text

            interpret_text = Linguistic_Module(self.chat_id)
            interpret_text.check_if_question_exists(str(self.message))

        else:
            error_message = "Messages must be in a text format, consisting of a SHERLOCK
question."
            print error_message
            self.bot.sendMessage(self.chat_id, error_message)
            return


    def run(self):
        while True:
            time.sleep(1)


class Visual_Module:

    def __init__(self, chat_id):
        """ Apply's the SIFT algoriothm for object classification based off
             training images. """

        self.current_directory = dirname(realpath(__file__))
        self.bot = telepot.Bot('353119581:AAGUpjclZ2RjWW-L-OrHU4bTHltqB1SGYFc')
        self.train_scene_direc = "scenes/"
        self.train_obj_direc = "objects/"
        self.scenes_direc = join(self.current_directory, self.train_scene_direc)
        self.scenes = [f for f in listdir(self.scenes_direc) if
isfile(join(self.scenes_direc, f))]
        self.chat_id = chat_id


    def find_scenes_with_objects(self, subject_data, object_data):
        """ Searches for Objects within scenes. """
        positive_scenes = []
        object_to_find = subject_data[0]
        maximum_distance = subject_data[1]
        minimum_matches = subject_data[2]
        input_image_name = subject_data[3]


        for scene in self.scenes:
            positive_scenes.append(self.test_image(self.train_scene_direc + scene,
self.train_obj_direc + object_to_find, minimum_matches, maximum_distance,
input_image_name))
```

```
        scenes_with_object = [i[0] for i in positive_scenes if i != None]

        if len(scenes_with_object) is 0:
            print "No Scenes found with object."
            self.bot.sendMessage(self.chat_id, "No Scenes found with object.")
        else:
            subject = self.find_objects_within_refined_scenes(scenes_with_object, 
object_data)
            return subject



    def find_objects_within_refined_scenes(self, scenes_with_object, object_data):
        """ Searches for objects within refined scenes. """
        answer_list = []

        for details in object_data:
            input_image = details[0]
            maximum_distance = details[1]
            minimum_matches = details[2]
            input_image_name = details[3]

            for scene in scenes_with_object:
                answer_list.append(self.test_image(scene, self.train_obj_direc + 
input_image,  minimum_matches, maximum_distance, input_image_name))

        answer = "\n".join(set([i[1] for i in answer_list if i != None]))

        if len(answer) is 0:
            print "No Scenes found with object."
            self.bot.sendMessage(self.chat_id, "No objects found in scene.")
        else:
            return answer


    def test_image(self, test, train, min_match, max_dist, input_image_name):
        """ Performs image classification with the SIFT descriptor """
        # opens query and train iamge in grayscale
        query_image = cv2.imread(test,0)
        train_image = cv2.imread(train,0)

        # Initiate SIFT detector
        sift = cv2.xfeatures2d.SIFT_create()

        # find the keypoints and descriptors with SIFT
        kp1, des1 = sift.detectAndCompute(query_image,None)
        kp2, des2 = sift.detectAndCompute(train_image,None)

        FLANN_INDEX_KDTREE = 0
        index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
        search_params = dict(checks = 50)

        flann = cv2.FlannBasedMatcher(index_params, search_params)

        matches = flann.knnMatch(des1,des2,k=2)

        return self.get_best_match(matches, test, train, max_dist, min_match, 
input_image_name)


    def get_best_match(self, matches, test, train, max_dist, min_match, 
input_image_name):
        """ Works out the best match based on training data. """

        # store all the good matches as per Lowe's ratio test.
        good = []
        for m,n in matches:
            if m.distance < 0.7*n.distance:
                good.append(m)


        if len(good) >= int(min_match):
            match_array = [matches.distance for matches in good]
            # sorts matches by size
            matches = sorted(match_array, key = lambda x:x)
            # gets average of 20 best matches
            distance = sum(matches[:20])/20

            if distance <= int(max_dist):
                scene_path = test
                image = input_image_name
                return scene_path, input_image_name


class Communincation_Module:

    def __init__(self):
        """ Performs system communication through the Requests API"""
        self.port = 8004


    def post_model(self):
        try:
            with open('sherlock.ce', 'r') as sherlock_model:
                model = sherlock_model.read()
            requests.post('http://localhost:' + str(self.port) + '/sentences', data =
model)
        except requests.exceptions.RequestException as e:
            error_message = "Error posting model to instance. Please check your
connection with the localhost CEServer.\n"
            print error_message
            print e
            sys.exit(0)


    def post_to_shared_kb(self, tellcard):
        try:
            requests.post("http://explorer.cenode.io:" + "6789" + "/sentences", data =
tellcard)
        except requests.exceptions.RequestException as e:
            print "Posting to shared knowledge base failed. Please check your connection
with the external instance at http://explorer.cenode.io.\n"
            print e


    def get_instances(self):
        try:
            response = requests.get('http://localhost:' + str(self.port) +
'/instances').json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get instances. Please check your connection with the
localhost CEServer.\n"
            print e
            os._exit(0)

    def get_concepts(self):
        try:
            response = requests.get('http://localhost:' + str(self.port) +
'/concepts').json()
```

```
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get concepts. Please check your connection with the
localhost CEServer.\n"
            print e
            os._exit(0)


    def get_model_concept_with_id(self, id):
        try:
            response = requests.get("http://localhost:" + str(self.port) + "/concept" +
"?id=" + str(id)).json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get concept with ID. Please check the concept ID exists and
your connection with the localhost CENode exists.\n"
            print e
            os._exit(0)


    def get_model_instance_with_id(self, id):
        try:
            response = requests.get("http://localhost:" + str(self.port) + "/instance" +
"?id=" + str(id)).json()
            return response
        except requests.exceptions.RequestException as e:
            print "Unable to get instance ID. Please check the instance ID exists and
your connection with the localhost CENode exists.\n"
            print e
            os._exit(0)


class Linguistic_Module:

    def __init__(self, chat_id):

        self.sherlock_model = Communincation_Module()
        self.instance_data = self.sherlock_model.get_instances()
        self.concept_data = self.sherlock_model.get_concepts()
        self.sift_parameters = self.get_sift_parameters()


        self.bot = telepot.Bot('353119581:AAGUpjclZ2RjWW-L-OrHU4bTHltqB1SGYFc')
        self.chat_id = chat_id
        # 'What colour hat is Elephant wearing?' type question
        self.question_group_1 = ["q4", "q7", "q24", "q41"]
        # "Where is the apple?"" type question
        self.question_group_2 = ["q6", "q8", "q25", "q47", "q48", "q53"]
        # 'Which character is in the emerald room?' type question
        self.question_group_3 = ["q9", "q13", "q37", "q45", "q28"]


    def get_sift_parameters(self):
        """ Gets SIFT parameters relating to game objects"""
        concepts = self.concept_data
        for concept_name in self.concept_data:
            if concept_name["name"] == "sift parameter set":
                concept_id = concept_name["id"]
                self.sift_parameters =
self.sherlock_model.get_model_concept_with_id(concept_id)
        return self.sift_parameters


    def check_if_question_exists(self, input_string):
        """ Take question and check if it exists in the model. If it exists, extract
```

```
             useful data. """
        input_string = input_string.lower()
        question_data = self.get_model_questions()
        found = False
        for question in question_data:
            if str(question["values"][0]["targetName"]).lower() == input_string:
                subjectType = question["relationships"][0]["targetConceptName"]
                subjectName = question["relationships"][0]["targetName"]
                relationship = question["values"][1]["targetName"]
                questionID = question["name"]
                found = True

        if found:
            self.bot.sendMessage(self.chat_id, "Valid question found, forwarding for
processing.")
            self.process_question_data(subjectType, subjectName, relationship,
questionID)
        else:
            print "Invalid Question"
            self.bot.sendMessage(self.chat_id, "Invalid question, please enter a valid
SHERLOCK question.")
            return


    def get_model_questions(self):
        """ gets the questions associated with the sherlock game. """
        question_data = []
        for instance in self.instance_data:
            if instance["conceptName"] == "question":

question_data.append(self.sherlock_model.get_model_instance_with_id(str(instance["id"]))
)

        return question_data



    def process_question_data(self, subjectType, subjectName, relationship, questionID):
        """ Searches the model for other useful data relating to the question.
            different question formats have a different method of retreiving data.
            These questions ids have been placed in groups in the constructor. """

        objectType = ""

        try:

            # 'What colour hat is Elephant wearing?' type question
            if questionID in self.question_group_1:
                objectType = relationship
                for concepts in self.concept_data:
                    if concepts["name"] == relationship:
                        subjectTypeID = concepts["id"]
                        subjectTypeConcept =
self.sherlock_model.get_model_concept_with_id(subjectTypeID)
                        relationship = subjectTypeConcept["relationships"][0]["label"]


            # "Where is the apple?"" type question
            elif questionID in self.question_group_2:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        concept_id = concepts["id"]

                subjectTypeConcept =
self.sherlock_model.get_model_concept_with_id(concept_id)
```

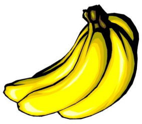
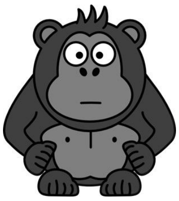
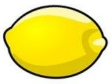
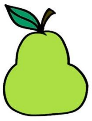
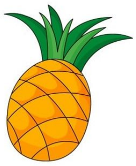
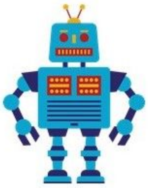
```
                for concepts in subjectTypeConcept["parents"]:
                    if concepts["name"] == "locatable thing":
                        concept_id = concepts["id"]

                subjectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(concept_id)

                for objectTypeValue in subjectTypeConcept["relationships"]:
                    locatableID = objectTypeValue["targetId"]

                objectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(locatableID)

                for objectTypeValue in objectTypeConcept["children"]:
                    objectType = objectTypeValue["name"]


            # 'Which character is in the emerald room?' type question
            elif questionID in self.question_group_3:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        subjectTypeID = concepts["id"]

                subjectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(subjectTypeID)

                for conceptValues in subjectTypeConcept["values"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        objectTypeID = conceptValues["targetId"]
                        break

                for conceptValues in subjectTypeConcept["relationships"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        objectTypeID = conceptValues["targetId"]
                        break

                objectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(objectTypeID)

                for objectTypeValue in objectTypeConcept["parents"]:
                    if objectTypeValue["name"] == "locatable thing":
                        locatableID = objectTypeValue["id"]

                objectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(locatableID)

                for objectTypeValue in objectTypeConcept["relationships"]:
                    relationship = objectTypeValue["label"]
                    locatableID = objectTypeValue["targetId"]

                objectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(locatableID)

                for objectTypeValue in objectTypeConcept["children"]:
                    subjectType = objectTypeValue["name"]


            # All other questions
            else:
                for concepts in self.concept_data:
                    if concepts["name"] == subjectType:
                        subjectTypeID = concepts["id"]
                subjectTypeConcept = 
self.sherlock_model.get_model_concept_with_id(subjectTypeID)

                for conceptValues in subjectTypeConcept["relationships"]:
                    if conceptValues["label"] == relationship:
                        objectType = conceptValues["targetName"]
                        break

            self.bot.sendMessage(self.chat_id, "We're looking for {0} associated with 
'{1}'".format(objectType, subjectName))
            print "We're looking for '{0}' associated with '{1}'".format(objectType, 
subjectName)

        except:
            print "Could not gather all data from the model"
            tb = traceback.format_exc()
            print tb
            return


        self.search_environment(objectType, subjectName, subjectType, relationship, 
questionID)


    def search_environment(self, objectType, subjectName, subjectType, relationship, 
questionID):
        """ Searches the environment of posters for objects"""

        subject_data = self.get_subjectName_image_path_and_params(subjectName)
        object_data = self.get_object_image_paths_and_params(objectType)

        image_recognition = Visual_Module(self.chat_id)
        found_object = image_recognition.find_scenes_with_objects(subject_data, 
object_data)


        self.construct_output(subjectType, subjectName, relationship, objectType, 
found_object, questionID)


    def get_subjectName_image_path_and_params(self, subjectName):
        """ Gets the subjectName image path and get training parameters
            for the image"""

        for index_entry in self.instance_data:
            if index_entry["name"] == subjectName:
                objectID = index_entry["id"]


        instance_id_data = self.sherlock_model.get_model_instance_with_id(objectID)

        subject_image_path = str(instance_id_data["values"][0]["targetName"])
        subject_image_name = str(instance_id_data["name"])

        for sift_obj in self.sift_parameters["instances"]:
            if str(sift_obj["name"]).lower() == subjectName.lower() + " parameters":
                siftID = sift_obj["id"]

        instance_id_data = self.sherlock_model.get_model_instance_with_id(siftID)

        for value in instance_id_data["values"]:
            if value["label"] == "maximum distance":
                max_distance = value["targetName"]
            elif value["label"] == "minimum matches":
                min_matches = value["targetName"]

        parameters = (subject_image_path, max_distance, min_matches, subject_image_name)
        return parameters


    def get_object_image_paths_and_params(self, objectType):
        """ Gets the objectType image paths and get training parameters
            for each of the images. """

        objectID_list = []
        object_name_list = []
        object_image_path = []
        object_image_name = []
        max_distance = []
        min_matches = []
        parameters = []

        for index_entry in self.instance_data:
            if index_entry["conceptName"] == objectType:
                objectID_list.append(index_entry["id"])
                object_name_list.append(index_entry["name"])

        for id_value in objectID_list:
            instance_id_data = self.sherlock_model.get_model_instance_with_id(id_value)
            object_image_path.append(str(instance_id_data["values"][0]["targetName"]))
            object_image_name.append(str(instance_id_data["name"]))

        for obj_name in object_name_list:
            for sift_obj in self.sift_parameters["instances"]:
                if str(sift_obj["name"]).lower() == obj_name.lower() + " parameters":
                    siftID = sift_obj["id"]

            instance_id_data = self.sherlock_model.get_model_instance_with_id(siftID)

            for value in instance_id_data["values"]:
                if value["label"] == "maximum distance":
                    max_distance.append(value["targetName"])
                elif value["label"] == "minimum matches":
                    min_matches.append(value["targetName"])

        for x, y, z, n in zip(object_image_path, max_distance, min_matches,
object_image_name):
            parameters.append((x,y,z,n))

        return parameters


    def construct_output(self, subjectType, subjectName, relationship, objectType,
found_object, questionID):
        """ Prints the output to the bot and to terminal """
        if questionID in self.question_group_1 or questionID in self.question_group_3:
            message_to_output = "The {3} \\'{4}\\' {2} the {0}
\\'{1}\\'".format(subjectType, subjectName, relationship, objectType, found_object)
            message_to_print = "The {3} '{4}' {2} the {0} '{1}'".format(subjectType,
subjectName, relationship, objectType, found_object)
        else:
            message_to_output = "The {0} \\'{1}\\' {2} the {3}
\\'{4}\\'".format(subjectType, subjectName, relationship, objectType, found_object)
            message_to_print = "The {0} '{1}' {2} the {3} '{4}'".format(subjectType,
subjectName, relationship, objectType, found_object)


        print "CNL: {0}".format(message_to_print)
        self.bot.sendMessage(self.chat_id, message_to_print)
        agentName = "sherlock"
        botName = "sherBot"
        tellcard = """there is a tell card named 'msg_{{uid}}' that has '{0}' as content and is to the agent '{1}' and is from the agent '{2}' and has the timestamp '{{now}}' as timestamp""".format(message_to_output, agentName, botName)

        self.sherlock_model.post_to_shared_kb(tellcard)

        final_message = "Observation posted to shared KB."
        print final_message
        self.bot.sendMessage(self.chat_id, final_message)


if __name__ == "__main__":
    Communincation_Module().post_model()
    go = telegram_handle()
    print "AI Player Launched!"
    go.run()
```

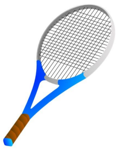
Appendix 6 - SHERLOCK Questions

Fruit related questions
What character eats pineapples?
What character eats apples?
What character eats bananas?
What character eats lemons?
What character eats oranges?
What fruit does Elephant eat?
What fruit does Leopard eat?
What fruit does Giraffe eat?
What fruit does Lion eat?
Sports related questions
What sport does Zebra play?
What sport does Lion play?
What sport does Giraffe play?
What sport does Hippopotamus play?
What sport does Elephant play?
What character plays rugby?
What character plays basketball?
What character plays soccer?
What character plays golf?
Location Questions
Where is the apple?
Where is the pear?
Where is Hippopotamus?
Where is Lion?
Where is Giraffe?
Where is Elephant?
What fruit is in the silver room?
Which character is in the emerald room?
What character is in the sapphire room?

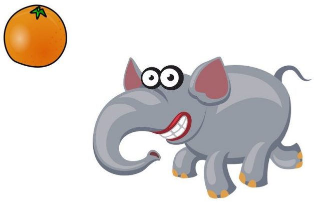
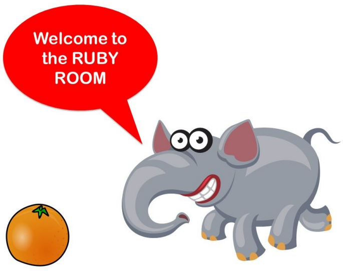
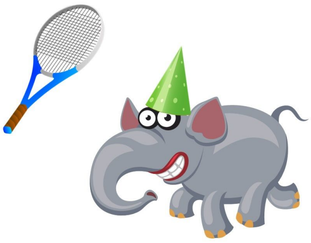
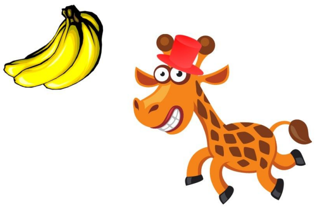
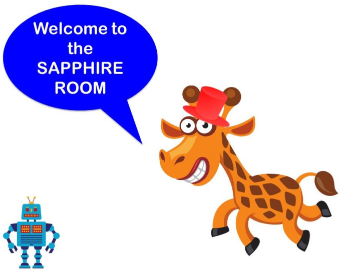
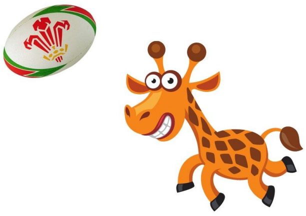
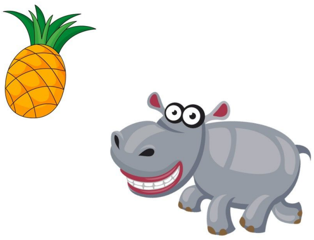
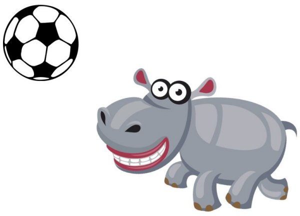
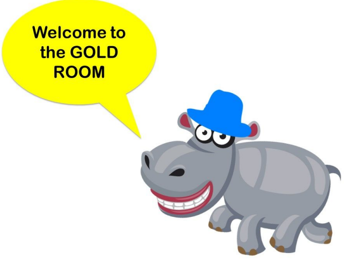
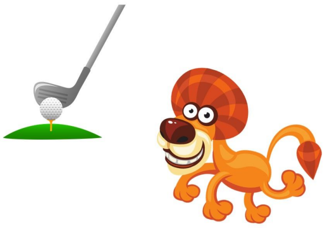
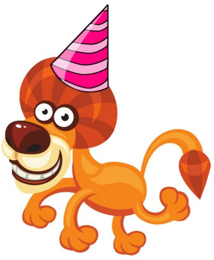
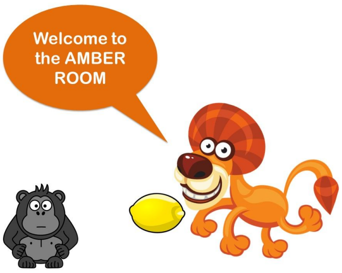
What character is in the ruby room?
What character is in the amber room?
Hat Questions
What colour hat is Elephant wearing?
What colour hat is Lion wearing?
What colour hat is Zebra wearing?
What colour hat is Hippopotamus wearing?
What character is wearing a yellow hat?
What character is wearing a blue hat?
What character is wearing a red hat?

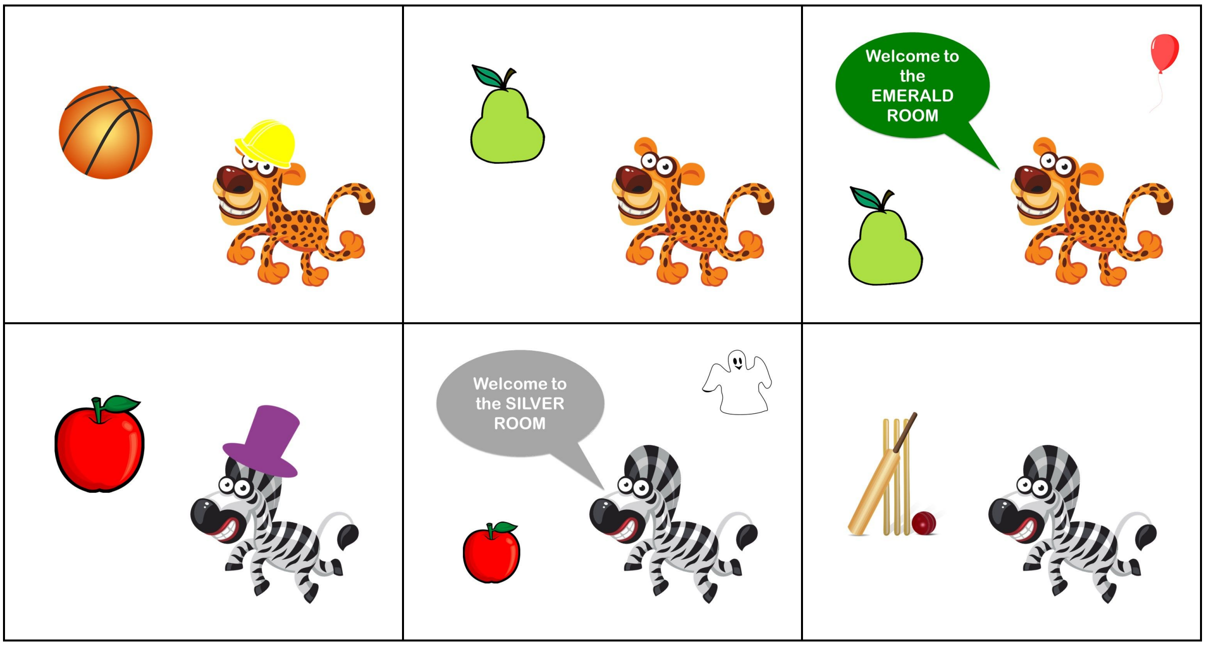
Appendix 7 - Objects

	Welcome to the SAPPHIRE ROOM	Welcome to the SILVER ROOM			

Appendix 8 - Scenes


Welcome to the EMERALD ROOM
Welcome to the SILVER ROOM