ELQA: Extensible Learning Question Answerer

COM3401 Stage III Report

Christopher Larcombe – cl234

Abstract

An question answering artificial intelligence system was designed, implemented and evaluated. The design is a symbolic, distributed, multi-agent system, in which several agents are used to operate independent components encapsulating specific functions. The design allows agents to collaborate, combining the function of different components to solve tasks of higher complexity requiring multiple functions. The system is supervised, i.e., trained by examples, allowing the recognition of associations between natural language messages and different combinations of components. Such associations are made autonomously and are not directly stipulated by the training examples, which contain only an example message and corresponding correct answer. A limited amount of testing is completed, but the implementation of the system is shown to function as expected, with few anomalies.

I certify that all material in this report which is not my own work has been identified.

. . . . . .

Christopher Larcombe

Contents

1 Introduction

In this document a question answering artificial intelligence system, ELQA, is presented and evaluated. The project was first considered based on the motivation to design a learning system, or *digital assistant*, capable of carrying out a variety of different tasks as instructed or requested to do so in natural language. As well as autonomously acquiring the necessary linguistic knowledge to interpret and usefully respond to arbitrary messages, a strong emphasis was placed on doing so by combining the functionality of several basic 'skills', each specialised to a specific task.

To a human being, it is intuitive to assume an 'intelligent' entity of any kind possessing specific skills should have the ability to apply those skills in a variety of different situations, and in combination with other skills if required to do so. After further consideration of the problem, it was a natural decision to separate each 'skill' in the system, distributing its functionality across a set of *components*, which could be easily combined. The concept of establishing associations between combinations of such components and the numerous patterns found in natural language messages soon formed the foundation of the project.

In its entirety the project represents an *approach* or starting point, which may be independently built upon, expanded, or integrated into a larger system. It is one of many possible paths that can be taken in exploring the problem of *question answering*.

1.1 Aim

The aim of the project was to develop a system possessing the following capabilities:

- Demonstrate several different functions as requested by natural language messages
- Demonstrate an ability to combine those functions arbitrarily
- Learn to do so by analysing examples messages and corresponding correct answers
- Demonstrate extensibility both in terms of functionality and an ability to use that functionality
- Establish associations between messages and component combinations

2 Specification

To design an artificial intelligence system capable of *inductively* learning and thus 'discovering' how to make use of a set of pre-programmed *components*, C, to accomplish tasks as requested in natural language *messages*, M. Each message $m \in M$ must contain a question or instructive sentence with a determined answer, reachable by manipulating information contained in m using the functionality provided by one or more components $c \in C$.

Each component $c_i^{(n)} \in C$ acting as a *black-box*, must encapsulate a specific hard-coded function, having $n \geq 1$ inputs and a single output under defined domains:

$$c_i^{(n)} : \vec{D}_1 \times \cdots \times \vec{D}_n \to O$$

where $c_i^{(n)}$ is the i^{th} component with n inputs, $\vec{D}_n$ is the domain of the n^{th} input of c_i and O is the domain of its output. Each individual component will map a vector of inputs to a single output.

The system must assume it is capable of accomplishing a requested task, and may attempt to utilise *multiple* components by 'connecting' them together, combining their functionality. An inductive learning process should take place, whereby example messages and corresponding correct answers are analysed in order to learn when each component should be used.

2.1 Assumptions

1. *One-to-one correspondence between question and answer*
   Each unique question or instruction presented to the system has a *single* correct answer.

2. *Sufficient information*
   Each question or instruction must contain the sufficient information necessary to respond correctly. For example, questions such as "What is my name?" require additional information.

3. *Deterministic answers* An answer to a question must always be the same; messages such as "Generate a random number between 1 and 100" are beyond the scope of this specification.

4. *Necessary skills*
   The system is only asked questions capable of being answered by the combination of skills it possesses. For example, if the system is only able to sum and multiply numbers it is expected to answer questions involving summation, multiplication and arbitrary combinations of the two, but is not expect to answer questions requiring division.

3 Design

3.1 Overview

In this section a formal design is presented. The design is a distributed, extensible *multi-agent* system, capable of approximating the assumed injective function $f : X \to Y$, mapping messages in a natural language X to corresponding answers Y, requiring the application of specific 'skills'. Each 'skill', required to answer a question, is provided by an independent *component* encapsulating a specific hard-coded function, analogous to a *black-box*.

System components are each operated by a unique agent and can be used in combination with other through collaboration between different agents. By collaborating with each other, agents are able to combine the functionality of different components in order to answer more complex questions requiring multiple 'skills'. A particular structure or combination of components used in this fashion is referred to as a *component composition* throughout this document.

The system is able to operate in two modes: *searching* and *parsing*. Searching takes place during training, discussed in §3.3, where each agent in the system attempts to establish its identity by analysing training examples in order to acquire a set of *views*, introduced in §3.2.4. During training, each agent also obtains additional information, outlined in §3.2.3, subsequently used to assess the quality of different component compositions and agent collaborations.

When adequate training has been completed, new messages are collectively parsed by the agents, as guided by the views established during training. This method, View-guided Recursive Descent Parsing, is discussed in §3.4.

3.2 System Constituents

3.2.1 Tokens

Tokens are containers of information used by the system to transfer small amounts of data between agents and components. Each token belongs to one of four domains: either *boolean*, *numeric*, *word* or *wordstring*, determining the type of information contained within the token and the different system components it can be used with.

The domain *word* consists of tokens storing 'words', which are alphanumeric strings additionally containing apostrophes and hyphens. The domain *numeral* consists of tokens storing numerical data and the *wordstring* domain consists of strings containing several words, which may separated by delimiters. Any unknown character, such as '?', will be classed as an individual word.

Messages such as "What is one add 2?", can be used by the system to generate populations of tokens of different types. For example, the message "What is one add 2?" can be used to generate the tokens listed in Figure 1 below:

boolean: {}
numeral: {2.0}
word: {'What', 'is', 'one', 'add', '2', '?'}
wordstring: {'What', 'is', 'one', 'add', '2', '?', 'What is', 'is one', 'one add', 'add 2', '2?', $\cdots$, 'What is one add 2', 'is one add 2?', 'What is one add 2?'}

Figure 1: **Example population of tokens**

As exemplified above, characters from a message can be used to form more than one type of token; the set of *word* tokens generated from a message will in fact be a subset of the set of *wordstring* tokens generated. Similarly, the set of *boolean* and *numeral* tokens generated will be a 'subset' of the set of the *word* tokens generated, though the content of such tokens will *not* be represented as strings.

As the number of words in a message increase, the total number of *wordstring* tokens possible to generate increase analogously to triangular numbers[1]. Therefore, if an input message contains w words and n numerals there are a total of $\frac{1}{2}(w^2 + w) + w + n$ tokens to generate.

3.2.2 Components

System *components* are independent entities possessing a distinct functionality. Each capable of carrying out a unique task, they provide the system with a set of atomic 'skills' that can be used in combination with one another.

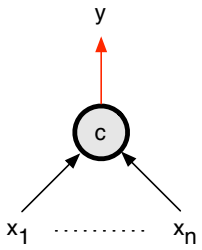

Figure 2: **A single component**

Although the functionality of each component may vary considerably, from the perspective of other system constituents each component is a fully encapsulated *black-box*, differing only in terms of input and output. Each component, see Figure 2, features one or more inputs and a single output. Each input and output is of a particular domain and can be used only with *tokens* compatible with that domain. When a component is executed with a vector of compatible tokens it may produce an output, in which case a new token of the appropriate domain is created to contain the output. If the component is executed and no output is produced, the component outputs a *null* value.

[1]The n^{th} triangular number can be calculated using the formula $\frac{1}{2}(n^2 + n)$

Components form *compositions* by simply using tokens output from other components as inputs. However, in order to do so, action is required by an *agent*. Each component is operated by its own agent, responsible for creating compatible *input vectors*.

3.2.3 Agents

System agents are responsible for operating components and collaborate with each other to 'join' different components together. Each agent is assigned a different component to operate and has the goal of creating new tokens. In order to execute its component an agent must obtain sufficient tokens to construct 'input vectors' compatible with its component. A compatible input vector is simply a particular ordered combination of tokens matching the domains of the inputs of an agent's component.

Each time a new token is created by an agent it is immediately stored in an *output vector*. Output vectors are passed between agents during parsing, storing information relating to the token they contain. The structure and function of output vectors in the context of parsing is discussed further in §3.4.1.

During training, each agent aims to learn when its participation is required in parsing messages by establishing a set of *views* V, introduced in the following section, §3.2.4. Additionally, each agent gathers data during training to aid the process of parsing messages requiring the participation of multiple agents.

An agent a can be formally defined as:

$$a = \langle V, [\rho, \epsilon], [\theta, \kappa], r \rangle$$

The probability ρ an agent's component will provide an output given a compatible input is estimated during training. The value is updated each time a component is executed by taking into account token output and the total number of times ϵ the component was executed.

Where b is an agent collaborating with a, *providing an input* to a, $\kappa_{a,b}$ is the total number of times a collaboration of this nature has been 'reinforced'. This value is subsequently used solely to calculate $\theta_{a,b}$, the percentage of such collaborations occurring out of the total reinforced collaborations between a and b in both 'directions', i.e.,

$$\theta_{a,b} \equiv \frac{\kappa_{a,b}}{\kappa_{a,b} + \kappa_{b,a}}$$

At any time, if there are no reinforced collaborations between an agent a and an agent b, then $\theta_{a,b} = \frac{1}{2}$, i.e.,

$$\kappa_{a,b} = 0 \wedge \kappa_{b,a} = 0 \rightarrow \theta_{a,b} = \frac{1}{2} \wedge \theta_{b,a} = \frac{1}{2}$$

The final element remaining, r, is the *short term memory* function. Each agent has a 'short term memory', which is used to recall recently parsed messages for efficiency.

3.2.4 Views

In the context of the design, a *view* refers to a particular way in which an agent may view all or part of a message to parse. More precisely, a view represents an input pattern familiar to an agent, providing it with a sense of identity and a method for locating and extracting information in messages corresponding to its component's inputs. The process by which views are formed and subsequently used is discussed in §3.3.1.

An agent a maintains a set of views V_a, where each view $\underline{v} \in V_a$ contains a vector of 'spaces' $\vec{s}$ consisting of an *input-space* i for every input of the agent's component and several interlacing *key-spaces* k. Similar to a regular expression, a view 'matches' a set of strings: each key-space k will directly match a string of *specific* words, while each input-space i can match an *arbitrary string* of one or more words. The key-spaces in a view therefore constitute a representation of the fixed part of a particular message format, while the location of the input-spaces indicate where variable parts, inputs, are likely to be found. Where n is the number of component inputs, $\vec{s} = [k_1, i_1, \cdots, k_n, i_n, k_{n+1}]$.

To give an example, if the system trains on a question such as "What is 2 add 3?", an agent responsible for a component capable of addition may form the view $[k_1 =$ 'What is ', $i_1, k_2 =$ ' add ', $i_2, k_3 =$ ' ?'], which can then be used to extract input from future questions in the same format. This process is referred to as *matching*, and in this example the input-spaces would 'capture' the sub-strings '2' and '3', making them available to the agent for further parsing. If a more complicated question was matched, such as "What is 2 times 2 add 1?", the input-spaces would capture '2 times 2' and '1'; input-spaces greedily capture *any* sub-string of a length greater than zero.

For the example view above, it is important to note that the inputs are commutative as the component associated with the view has commutative inputs. As this is not always the case, a mapping is required between each input-space and component input to ensure the agent refers to the correct input during and after parsing. For simplicity, each input-space $i \in \underline{v}$ can be assumed to hold an integer equal to the index of the component input it refers to.

When an agent uses a view to match a message, a *match result* $(\vec{m}, g, p)$ is returned, where $\vec{m}$ is a vector of captured sub-strings, g is the sum of the length of every key-space in the view and p is the percentage of characters in the message that matched the view. The g and p values reflect the quality of the match, which is further in §3.4.1.

Before a view can be used by an agent parsing a message it must first be verified and 'activated'. A view's activation status is given by the boolean activation indicator $z \in \underline{v}$, where $z \in \{0, 1\}$. To verify a view in order to activate it, several training examples successfully making use of the view are required. Such examples, composed of a message and corresponding correct answer, are permanently stored in $E \in \underline{v}$, a set of matching examples. When the total number of examples exceeds the system's activation constant α the view is activated, i.e., $z = 1 \leftrightarrow |E| \geq \alpha$. View activation is discussed in greater depth in §3.3.2.

In summary, a view $\underline{v}$ can be formally defined as:

$$\underline{v} = \langle \vec{s}, E, z \rangle$$

where $\vec{s}$ is a vector of 'spaces', E is a set of successfully matched messages with corresponding answers, and z is an indicator of the view's activation status.

3.3 Training and View Acquisition

In order for agents in the system to discover their identity and recognise when they are required, they must establish a familiarity with particular formats of messages requiring their involvement. By acquiring and using *views*, previuosly introduced in §3.2.4, an agent can assess when it is appropriate to participate in response to a message. The goal of training is to correctly establish each agent's views for use in *View-guided Recursive Descent Parsing*, described in §3.4.

Consider a set of training examples E, such that a specific example $e \in E = \{m \in M, r \in R\}$, where m is an *example message* corresponding to a *correct* response r. The system must correctly map as many instances of M to instances of R as possible. It is important to note that the component composition required to map m to r is not stipulated by the training example e; the system must autonomously 'discover' the composition required to do so, as stated in the specification. Unfortunately however, the number of possible compositions involving even a few components is beyond what can tractably be enumerated in search of a composition to reach r from m.

However, enumeration of all compositions consisting of one component only *can* be carried out without encountering a combinatorial explosion. In this situation, if an agent outputs a token equivalent to the answer r it has the opportunity to learn the structure of the message m and can begin to establish its identity, forming views. This method is adopted in the design, so has the limitation that training examples for the purpose of view acquisition are assumed to involve *one component only*. However, when several views have been established, more complex training examples involving multiple components can be used. The details of the search algorithm and how views are formed is discussed in §3.3.1.

The enumeration, or *searching* process, is the most important element of the training algorithm, but alone is insufficient to fully train the system. Without the help of other mechanisms, several inherent problems would occur, including *explosive* formation of a large number of superfluous views, and the incorrect formation of views by agents based on correct answers found purely by coincidence. In order to address these problems, the algorithm features two main safeguards:

1. Agents additionally use training examples to *verify* their existing views.
2. Agents only search if the answer cannot first be found by parsing the message.

The first safeguard listed above, *verification*, involves an agent *independently* parsing an example message without the help of other agents. During this parsing procedure a number of

the parsing agent's views may be *activated* or deleted, later discussed in §3.3.2. Verification also has a second purpose: by the second safeguard, if the agent is successful in parsing the message during verification, i.e., parsing the message results in the correct output being given, then a search is unnecessary as the agent is already capable of producing the correct answer.

If each agent proceeded to search regardless of its existing ability to correctly parse the message, a large number of superfluous views could potentially form. For example, consider an agent operating a component with several *wordstring* inputs and a *boolean* output, such as a component to test if a string contains a particular sub-string. Acknowledging each word or word-string in a message cannot be used more than once simultaneously, i.e., with more than one input, *every training example* with the correct answer 'False' would lead the search algorithm to discover the 'correct answer' with almost every input vector enumerated; most word-strings in the message would *not* contain other word-strings found in the message. Every time an input vector yielded false a superfluous view would form as a result, except in the one case where the correct word-strings were used. It is unavoidable, but acceptable that a certain amount of superfluous views may form the *first time* a component such as the one exemplified is trained. Most importantly, the second safeguard prevents superfluous views forming beyond the first training example. Any that do form during the first are not likely to become *activated* due to the first safeguard, and will simply be ignored by the agents.

It is important to emphasise that in the training algorithm, whether or not an agent searches is determined by its own ability to parse the message, not the system as a whole. By giving each agent the *opportunity* to search, independent of the success of other agents finding the answer, each agent has an equal chance of forming a correct view. If this were not the case, an agent could by coincidence reach the correct answer and thus subsequently prevent the correct agent from searching, reaching the correct answer and forming the correct view.

Once each agent has verified a training example and searched if necessary, the *system* attempts to parse the message using View-guided Recursive Descent Parsing. Several answers are likely to be returned, each with a corresponding *quality* value indicating their probability of correctness. If the correct answer is found among those returned and has the highest associated quality value then the system trained successfully, as it is this answer that would be returned to a user. If, however, an incorrect answer had the highest quality value, or the correct answer was found among incorrect answers of equal quality, the correct answer requires 'reinforcement' by the training algorithm. The quality of the composition providing the correct answer must be increased.

A situation such as this is likely to occur with messages requiring the participation of multiple agents. For example, disambiguation of messages, such as "What is 2 times 2 add 1?", require such training examples as they have multiple answers; in this case '5' or '6'. To reinforce the correct answer, '5', the structure of the composition providing the correct answer is reinforced by modifying the κ values corresponding to each unique pair of collaborating agents in the composition. For every agent a' receiving an output vector from agent a, the value $\kappa_{a',a}$ is incremented by 1, signifying a correct collaboration between agent a' and a.

It is also possible that the answer may not be among those returned by the parsing algorithm. When this is the case, no further action is taken as the system requires further training examples to activate any existing views associated with the message.

3.3.1 Searching

When presented with a training example e, the system first generates four populations of tokens from the message $m \in e$: one for each of the token domains, containing all possible tokens of that domain able to be generated from the message (See Figure 1 on page 5 for an example with the question "What is one add 2?"). For each token t generated, the start and end 'location' $L_t = (l_1, l_2)$ in the message where the data contained in t was extracted is stored. Each location starts at zero, incrementing with each word and delimiter. For example, the location of the *numeric* token '4' taken from the message "What is the square root of 4?" is $(12, 12)$ and the location of the *wordstring* token 'What is the' is $(0, 4)$. As there is no delimiter between the '4' and '?' in the message, the location of '?' is $(13, 13)$. These values are important when generating views, as they allow the static parts of the message to be identified that were not used to form tokens for component input.

In the next stage of the search procedure each agent searching uses the token populations to enumerate every possible input vector compatible with their component in search of the answer, which is already known to each agent. Input combinations are excluded from the search if more than one token contains the same data extracted from the message; each character in the message can only be used once. If the locations of any two tokens in an input vector overlap, the input vector is abandoned and not used by the agent. It is also at this stage that the commutativity of an agent's component is taken into account. If the agent has a component with commutative inputs, only input vectors involving a unique combination of tokens are used to execute the component.

Each time an agent executes its component while searching, the value ϵ_a keeping track of the total number of component executions for that agent, a, is incremented:

$$\epsilon_a = \epsilon'_a + 1$$

where ϵ'_a is the total before the update.

Subsequently, an estimate of the probability ρ_a that the component will provide a new token given a compatible input vector is also updated. If a new token was output from the component, the following update is used:

$$\rho_a = \frac{\rho'_a \epsilon'_a + 1}{\epsilon'_a + 1}$$

If, however, the component did not output a new token, the following alternative update is used:

$$\rho_a = \frac{\rho'_a \epsilon'_a}{\epsilon'_a + 1}$$

If the agent executes its component on an input vector resulting in the creation of a new token equivalent to the known answer, a view are formed based on the input vector and the message. Views can be formed using an algorithm such as Algorithm 3.1, shown below:

Algorithm 3.1: FORMVIEW(v, m, n)

comment: form a view $\underline{v}$ based on input-vector v, message m and answer n

$b \leftarrow \lambda$
$\vec{s} \leftarrow (\,)$
for each word or delimiter w at location i in message m
do
- **for each** token t in v corresponding to input j
  - **do**
    - **comment:** If the location of w is in the location of t
    - **if** $i \geq L_{t,1}$ **and** $i \leq L_{t,2}$
      - **then**
        - **comment:** If b is not empty
        - **if** $b \neq \lambda$
          - **then**
            - **comment:** Add a new key-space and clear buffer
            - $\vec{s} \leftarrow \vec{s} + b$
            - $b \leftarrow \lambda$
        - **if** $i = L_{t,2}$
          - **then**
            - **comment:** Add a new input-space
            - $\vec{s} \leftarrow \vec{s} + j$
        - **break**
  - **else** $b \leftarrow b + w$

$\underline{v} \leftarrow (\vec{s}, \{< m, a >\}, 0)$
return $(\underline{v})$

Following the production of the new view $\underline{v}$, the agents check its set of view V_a before adding $\underline{v}$ to it to ensure no duplicate views are stored.

3.3.2 View Activation and Generalisation

During training and view acquisition, certain training examples may cause *multiple* agents to form views. As each training example is assumed to involve only one component and each is operated by a different agent, only one agent is expected to be capable of finding the answer and subsequently form a view. However, for certain training examples the inputs may be such that an unrelated agent reaches the same answer by 'coincidence'; this is likely occur more often with training examples with commonly occurring answers, such as 'True' or 'False'.

To give a more unusual but obvious example, in a system containing an addition and multiplication component operated by two untrained agents, the answer to the question "What is 2 times 2?" can be reached by both summing and multiplying the numbers in the message. As this is the case, *both* agents will reach the correct answer during enumerative search, and both subsequently form a view. Having been exposed to this bad example, the agent operating the addition component will have formed an incorrect view containing the word 'times' in its view's key-space. If the addition agent was immediately able to use this incorrect view, it would answer multiplication questions as if they were addition questions.

In order to prevent this problem occurring, a view is not able to be used by an agent until several training examples have confirmed it is correct. As previously mentioned in §3.2.4, several training examples successfully making use of the view are required to *activate* it. When a view is created, or used to generate a correct output during the verification phase of training, the message and corresponding answer is stored. Each time a new example is added, the total number of examples is checked. If this number exceeds α, the system's activation threshold, the view is then activated. Additionally, in the verification phase if a view fully matches a message ($p = 1$) and is subsequently used to generate an input vector which upon execution outputs an incorrect answer, the view responsible is deleted. However, views containing no key-spaces are exempt and not deleted.

Each time a view is activated, it is intersected with every other activated view in order to form *generalised views*, see Figure 3 below:

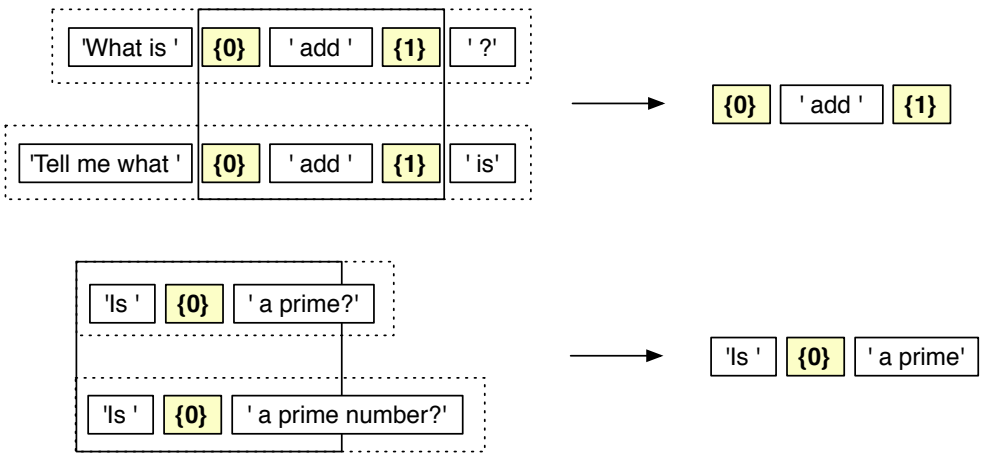

Figure 3: **Two example view intersections, or generalisations**

A generalised view can only be obtained from two views if the two views compared share the same input-spaces in they same order, and are separated by equal key-spaces. If the views being intersected both have space vectors beginning with a key-space and one or more characters of which from the *right hand side* to the left of the key-space are equal, the new view will include a key-space containing the sub-string of characters in common with the two key-spaces. Similarly, at the end of the two views, if both share a key-space beginning with one or more of the same characters, the matching characters will form a key-space at

the end of the space vector in the new view. However, key-spaces in the new view containing only a single space are not added to the space vector at the beginning or end of the space vector. Furthermore, if the new view formation is complete and the space vector contains no key-spaces, it is abandoned and will not be used.

If a new generalised view is successfully created and does not already exist, it is activated and added to the agent's set of views.

3.4 View-guided Recursive Descent Parsing

Once the system has adequately trained and all relevant views have been established, the agents are ready to collaborate with each other in order to parse messages. In this section a recursive method for constructing useful component compositions is presented. Compositions are recursively constructed in a 'top-down' fashion, beginning with the component that will output the final answer and ending with components taking input directly from messages.

When the *system* receives a new message to parse from a user, that user is seeking the same response the system aims to find within a token held in an 'output vector', to be returned by an agent. Upon receiving a new message, each agent in the system is requested to parse it. An agent's parsing procedure can be broken down into two stages, which are repeated *for each of the agent's views*, before a final set of output vectors containing tokens are returned:

1. Obtaining 'output vectors' to construct 'input vectors'
2. Produce new tokens in output vectors to return by executing the component using constructed input vectors

When an agent receives a message to parse, it first attempts to match it against its activated views in order to locate and extract any information within the message able to be used directly as an input to its component. If a view matches a message, a sub-string for each input-space in the matching view will be available for the agent to analyse. As each input-space is mapped to a component input by the view, each sub-string holds the same relationship and can be used with an input if a compatible token is able to be constructed from it.

If a sub-string can be used to construct a token compatible with the input it is mapped to by the view, it is used to create an output vector and stored for later use with that input. After this check has been made, regardless of the result, the sub-string is then sent to all agents in the system for parsing with an output domain equal to the domain of the mapped input. It is at this point where the recursion begins. Each agent parsing the sub-string will return a set of output vectors to the agent, or an empty list if unsuccessful, which will again be stored for later use with the mapped component input.

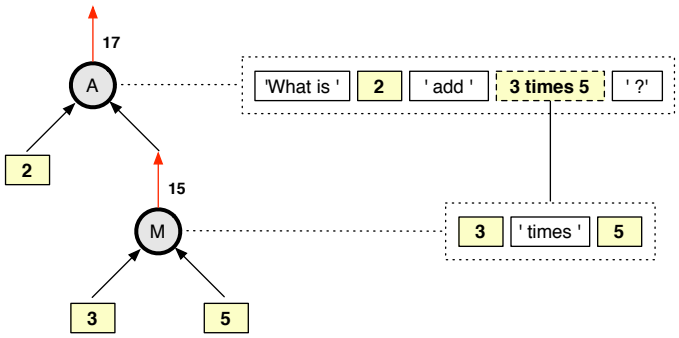

Figure 4: **An example parse**

After the agent has gone through this process with each input-space of the matching view, a set of output vectors for each of the component's inputs should be available. If the agent was unable to find any output vectors for one or more of its component's inputs, the component cannot be executed and the process begins again with the next view. If there are sufficient output vectors available, i.e., at least one for each component input, the tokens held within them are used to construct all possible combinations of valid input vectors for executing with the component.

It is important to note that each output vector directly corresponds to a unique component composition, possessing a unique 'history' describing its structure. When the component is executed on an input vector, a new composition is effectively being 'tested'; a composition of higher complexity, *uniting* the unique history of each input. If the component produces an output, the new composition will continue its existence as an output vector until it is used to form part of a more elaborate composition in the same manor it was created. However, if the agent operating the component was called by the system and not by another agent, upon being returned, the composition will be considered a candidate solution to the original message parsed.

Ideally, when all agents have finished parsing the original message the system should have an output vector containing the correct answer. At the highest level of the recursion, where each agent returns set of output vectors to the system, it is highly likely that more than one output vector in total will be returned. The system must therefore decide which is the 'best' answer to return to the user. In order to make this important decision, each answer, or composition, is assigned a *quality value*.

3.4.1 Composition Quality

A *quality value* q is assigned to each output vector $\vec{o}$, such that the larger the value of q, the more 'preferable' a token $y \in \vec{o}$ is to the system. The structure of each output vector $\vec{o}$ can be formally defined as:

$$\vec{o} = (q, v, \vec{t}, y)$$

where q is a quality value, y is a token, v is the total number of views used in the creation of y, and $\vec{t}$ is a 'trace' of output vectors used to construct the token, describing the structure of

the composition that produced y. When an output vector is produced based on a token containing data taken directly from a message, the output vector is of the form $(0, 0, (\emptyset, ()), y)$ and the quality value is zero.

In order to make certain assessments requiring knowledge of the composition structure, $\vec{t}$ is referenced. Where a is an agent and $\vec{x}$ is a *vector of output vectors* from which tokens were extracted and used to generate y:

$$\vec{t} = (a, \{\vec{t'} : \forall \vec{y} \in \vec{x}(\vec{t'} = \vec{y}_3)\})$$

When a new output vector $\vec{o}$ is produced following execution of a component on *a vector of output vectors* $\vec{x}$, the value $v \in \vec{o}$ is calculated by summing the previous output vectors v values and adding one:

$$v = \Big(\sum_{\vec{o'} \in \vec{x}} \vec{o'}_2\Big) + 1 \tag{1}$$

A quality value is calculated is calculated each time this occurs. Where a is an agent constructing a new output vector $\vec{o}$ following the execution of its component, again on an input vector derived from *a vector of output vectors* $\vec{x}$, the new quality value $q \in \vec{o}$ is defined as:

$$q = \Big[p(g + \beta\vartheta(\vec{t}, 0)) + \eta(1 - (\rho_a + \frac{1}{2}))\Big] + \sum_{\vec{o'} \in \vec{x}} \vec{o'_1} \tag{2}$$

The value depends on the sum total $\sum_{\vec{o'} \in \vec{x}} \vec{o'_1}$ of the quality of the output vectors in $\vec{x}$ and several other factors, some of which are more subtle than others. The most significant factor is the total number of key-space characters matched: the sum of the length of every key-space of the view used to construct the composition, denoted g. For example, $g = 14$ for the simple composition required to answer the question "What is 2 add 3?" using the view $[\text{'What is '}, i_1, \text{' add '}, i_2, \text{' ?'}]$. Other views with space vectors such as $[i_1]$ or $[\text{'What is '}, i_1, \text{' ?'}]$ will also match the message, but with lower g values: 0 and 10 respectively. The value gives an indication of the degree of generality to which a view matches a message; views with less key-space characters can potentially match a larger number of possible messages.

The second most significant factor p is the percentage of the message matched by the view, and is multiplied by g. This value decreases the significance of the update proportionally to the amount of the message the view matched. This allows messages such as "So what is 2 add 3?" to be matched, but at a cost.

The value $\beta\vartheta(\vec{t}, 0)$ in equation (2) above is used discriminate between compositions that would otherwise have equal qualities, commonly arising from ambiguous messages such as "What is 2 times 3 add 1?". In order to disambiguate the message, the parsing procedure must recognise that certain combinations of agent collaboration are preferable over others. Due to the rule of *order of operations* relevant to this example, the multiplication 2×2 must be carried out before the addition, which in terms of component compositions translates to the agent responsible for addition 'preferring' to receive an output from the agent

responsible for multiplication, rather than the opposite way around, i.e., the multiplication agent receiving an output from the addition agent. As previously introduced in §3.2.3 and discussed further in §3.3, each agent can be trained to prefer sending or receiving output vectors to and from other agents by modification of their κ values.

The constant value β is a weighting, or parameter, used to decrease the effect the value of the function $\vartheta(\vec{t}, 0)$ has on the composition quality. The value of β should be kept relatively small to ensure collaboration preference only becomes significant when necessary. If the value is set too large the system may choose to return the result of an incorrect composition on the basis it contained preferable collaborations, failing to take into account the greater significance of other the properties of the composition, such as p and g, previously discussed. The function ϑ uses the trace $\vec{t} \in \vec{x}$ to analyse the composition structure, considering each collaborating agent's θ value. The function ϑ is defined as:

$$\vartheta(\vec{t}, \lambda) = \frac{\sum_{\vec{t'} \in \vec{t}_2 \wedge |\vec{t'}_{2,2}| > 0} \left(\theta_{\vec{t}_1, \vec{t'}_1} - \frac{1}{2} + \vartheta(\vec{t'}, \lambda + 1) \right) + \sum_{\vec{t'} \in \vec{t}_2 \wedge |\vec{t'}_{2,2}| = 0} \left(\theta_{\vec{t}_1, \vec{t'}_1} - \frac{1}{2} \right)}{2^{\lambda} |\{\vec{t'} : \vec{t'} \in \vec{t}_2, |\vec{t'}_{2,2}| > 0\}|} \tag{3}$$

The function works by averaging the value $\theta_{a,b} - \frac{1}{2}$ of each collaborating agent a and b in the composition, decaying each average by $\frac{1}{2^{\lambda}}$ where λ is the 'depth' of the collaboration in the composition, i.e., the number of agent collaborations between a and the agent at the 'top' of the composition giving the final output. As $\theta_{a,b}$ is initialised to $\frac{1}{2}$ if a and b have no reinforced collaborations, see §3.2.3, a value of zero is ensured in this situation as $\frac{1}{2}$ is subtracted from $\theta_{a,b}$ in the equation. This of course applies to the whole equation, which will return zero if no collaboration data is available.

The penultimate factor affecting composition quality as defined in equation (2) is ρ, the probability a component will output a new token given an input vector, as previously introduced in §3.2.3 and elaborated on in §3.3.1. Controlled by a constant weight η, where $\eta < \beta$, the value is 'deciding factor' if all the composition properties previously introduced are equal. Though not vastly important to the performance of the system, some situations necessitate this level of discrimination. For example, two different components operated by two agents: a component to reverse strings and a component to look up the capital cities of countries. If each agent possessed a view containing the input space $[i_1]$, without the need for collaboration with other agents, both would successfully demonstrate their functionality when presented with *any* message to parse. i.e., when requested to parse the string "France", the system would receive two output vectors: one containing 'ecnarF' and the other containing 'Paris'.

In order to decide which of these tokens should be returned to the user, ρ is taken into account and the output from agent operating the component with the lowest token output probability, the smallest ρ value, is chosen. If both components had an equal ρ value in this situation, the quality values would be equal and the system would have no choice but to return both answers to the user.

The final factor affecting composition quality is the complexity of the composition, or number

of views used to construct it. This final assessment is not made by the agents and therefore does not feature in equation (2). Rather, the complexity of the composition is taken into account by the system when comparing the qualities of the output vectors offering candidate solutions, i.e., output vectors returned to the system by agents called to parse the original message. An averaged quality value $\overline{q}$ is calculated as follows:

$$\overline{q} = qv^{\gamma-1} \tag{4}$$

Where $\vec{o}$ is an output vector offering a possible answer and $q, v \in \vec{o}$. The constant γ varies the 'strength' of the average, and should be set to approximately 0.8. If this value is set too high, e.g., if $\gamma \geq 1$, then the most significant factor, the total key-space characters, is not averaged and superfluous compositions will be scored equally even preferred to more simple compositions solving the problem; if an average is not taken, there can be a high number of components in the composition with a low quality. For example, a correct composition containing one component providing the capital city of a country will not be preferred over the same composition with a component that reverses strings attached, resulting in the reversal of the string containing the capital city. If the average is too strong however, complex compositions involving several steps that can only be made using low quality components may not be chosen.

3.4.2 Avoiding Loops and Superfluous Compositions

During parsing, each an agent will extract several sub-strings from the messages they receive to parse, which they will recursively request to be parsed by compatible agents. In situations where the sub-string extracted is equal to the same message, a loop is possible if the input and output domains allow. For example, an agent possessing the view $[i_1]$, operating a component to reverse a string, which will match any message, calling itself infinitely; see Figure 5 below:

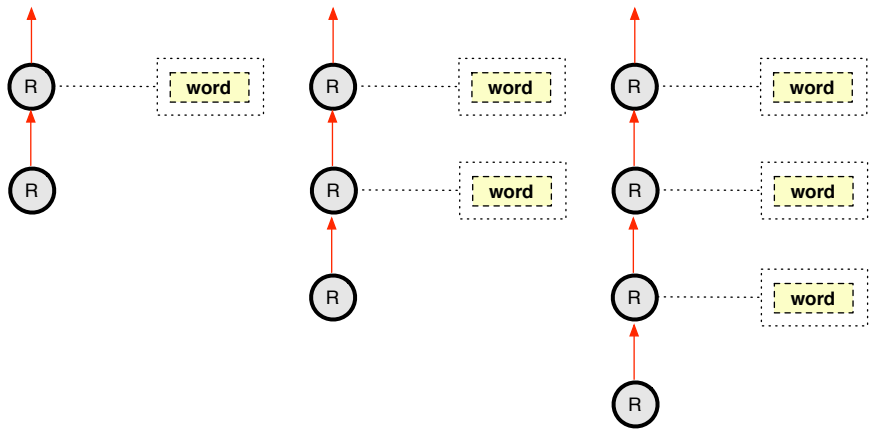

Figure 5: **An agent is infinitely called to parse the same message**

In this situation no strings are ever reversed as the composition is never completed. In order to avoid this kind of loop, each time an agent receives a message to parse it stores a record

in a *history*; i.e., each time agent a parses a message m, the history set H is updated:

$$H = H \cup \{\langle a, m \rangle\}$$

This history set checked before calling another agent to ensure it has not previously received the message it is about to be requested to parse. Each time an agent is called to parse a message by another agent, a history set is also passed to the agent being called.

As well as the history set, a trace of token content T is appended to each time an agent is able to use a message sub-string directly with its input. For a particular component input i, if a sub-string s can be used to form a token compatible with i, each agent subsequently called to parse s will be sent the trace $T \cup \{s\}$. If the same substring is not able to be used immediately as an input, each agent subsequently called is sent the trace T. Thus, each time an agent executes its component and produces an output in T, the output vector does not need to be returned as it would represent an inefficient path to an input already 'used'.

3.4.3 Short Term Memory

To increase efficiency, each agent a uses a short term memory function r_a, mapping messages m to tuples containing a set of output vectors O and a token trace T. When an agent returns a set of output vectors, a tuple is added to the function, i.e., $r_a(m) \cup \{\langle T, O \rangle\}$.

Each time an agent receives a new message to parse, it is able to check these tuples and avoid parsing a message it has parsed before. It is important to note that the history H does not prevent this from occurring as the short term memory is temporally persistent, only being reset once the *system* has finished parsing the message. Each agent in turn attempts to parse the full message, so there are different histories throughout the parsing; one for each agent.

A token trace is stored because it potentially limits the output vectors generated while parsing. If a remembered output vector set and message combination had an associated token trace, which was not a subset of a new token trace associated with a new message to parse, the remembered output vector set may be a subset of the output vector set obtainable by parsing, i.e., despite recollection of the new message, if the token trace stored is 'larger' than the one associated with the new message it cannot be guaranteed to contain all the obtainable output vectors. If this is the case, the memory is not used and parsing continues normally.

If the opposite is true, and the new token trace is 'larger' than the stored one, i.e., containing it and additional tokens, the memory can be used. The extra tokens in the new token trace are simply used to exclude output vectors that used equivalent tokens in their production.

4 Implementation

In order to verify the design and test its effectiveness, an implementation of the design was developed in the high-level object-oriented programming language Python. This programming language was chosen for several reasons, including the simple syntax and high-level data structures. As the implementation is only a proof of concept, potential GUI development or code execution speed was not considered to be relevant.

In addition to the ELQA system itself, a simple extensible *shell* was written to make certain tasks easier. The shell features a built in help system, and can be used to train, test and reset the system, display data stored in agents and set system parameters. The test and train functions directly call functions with the system, which can operate on single training examples or files containing several training examples. Such files contain links to other files, and are parsed recursively. In addition to the shell, a small class called *TextStyler* was also implemented to colour console text on UNIX based systems, making debugging easier.

Excluding individual components, the system consists of 8 main classes; *ElqaSystem*, *ElqaConfiguration*, *Agent*, *Token*, *Component*, *View*, *InputSpace* and *KeySpace*. Each class maps very directly to the design presented in this document.

Data is stored using the Python *Pickle* module; each agent is stored in a *.dat* file, as are the configuration settings. When the system is initialised upon instantiation of *ElqaSystem*, the folder 'Components' in the directory structure is scanned and a list of components is generated. If an agent data file exists for a scanned component, it is read in directly using Pickle and initialised. If a component is detected and does not have an associated agent data file, a new agent is created for the component. This design allows a high level of extensibility, making it very easy to add new components; modification of the core code is not necessary. The *Agent* class stores all information gained during training; each agent is saved to disk when the system exists.

4.1 Classes

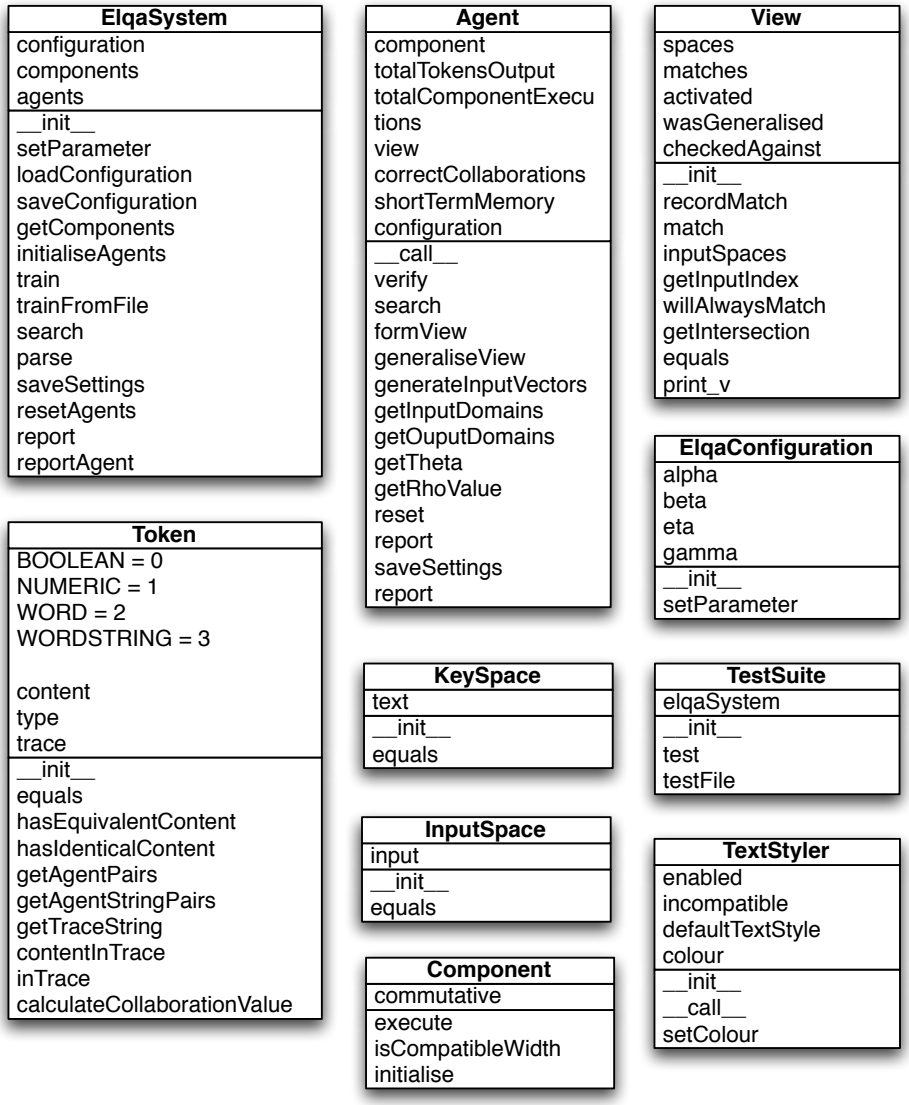

Figure 6: **Class diagram**

5 Demonstration

In this section a fully operational Python implementation of the ELQA System is demonstrated, based on 17 different components created for testing purposes. Various examples messages were parsed using the *ELQA Shell*, see Figure 7 below, and have been transcribed for presentation in this document.

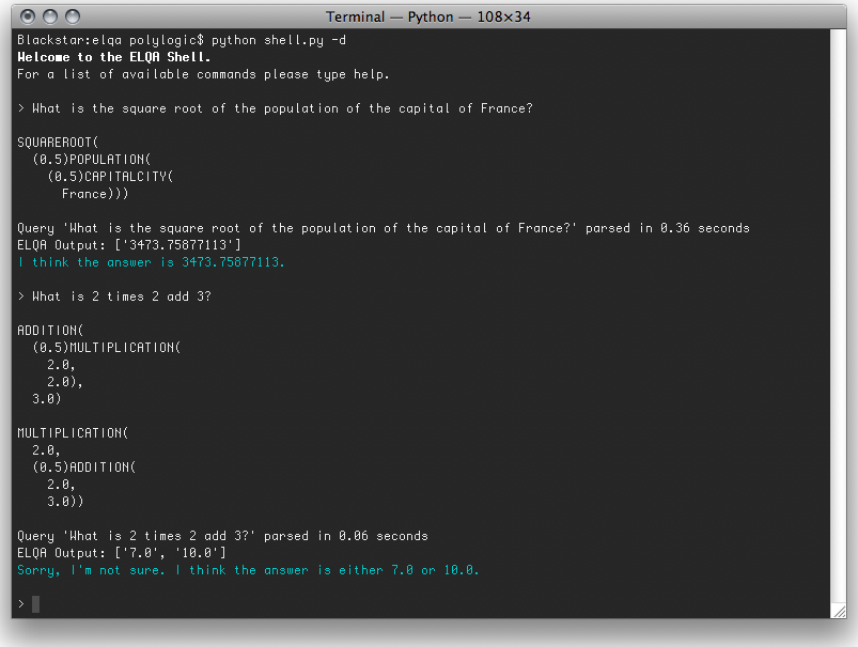

Figure 7: **Parsing messages in the *ELQA Shell***

5.1 Example Compositions

The two transcripts directly below demonstrate how the system is able to parse simple messages phrased in different ways, requiring the use of a single component. The system is able to parse "Now multiply 10 and 100" by ignoring the word "Now" at the beginning of the message.

Addition

```
> What is 12 add 21?

ADDITION(
  12.0,
  21.0)

Query "What is 12 add 21?"
-> ['33.0'] returned in 0.09 seconds
I think the answer is 33.0.

> Add 37263 and 5937 together

ADDITION(
  37263.0,
  5937.0)

Query "Add 37263 and 5937 together"
-> ['43200.0'] returned in 0.11 seconds
I think the answer is 43200.0.
```

Multiplication

```
> What is 7 times 5?

MULTIPLICATION(
  7.0,
  5.0)

Query "What is 7 times 5?"
-> ['35.0'] returned in 0.14 seconds
I think the answer is 35.0.

> Now multiply 10 and 100

MULTIPLICATION(
  10.0,
  100.0)

Query "Now multiply 10 and 100"
-> ['1000.0'] returned in 0.08 seconds
I think the answer is 1000.0.
```

The example transcript below demonstrates a more complex composition, composed of two collaborating agents. The system is able to reach the correct despite the ambiguity in the question, as the appropriate collaboration between the *addition agent* and the *multiplication agent* was reinforced during training.

Combining addition and multiplication

```
> What is 2 add 3 times 4 times 5 add 6 add 7 times 8?

ADDITION(
  (0.5)ADDITION(
    (0.5)ADDITION(
      2.0,
      (1.0)MULTIPLICATION(
        (0.5)MULTIPLICATION(
          3.0,
          4.0),
        5.0)),
    6.0),
  (1.0)MULTIPLICATION(
    7.0,
    8.0))

Query "What is 2 add 3 times 4 times 5 add 6 add 7 times 8?"
-> ['124.0'] returned in 0.86 seconds
I think the answer is 124.0.
```

Below are further example transcripts demonstrating several more complex compositions involving several components of different domains.

Multiple Components

```
> What is the square root of 2 miles in inches add the population of the capital of France times TWO?

ADDITION(
  (1.0)SQUAREROOT(
    (0.5)CONVERTER(
      miles,
      inches,
      2.0)),
  (1.0)MULTIPLICATION(
    (0.5)POPULATION(
      (0.5)CAPITALCITY(
        France)),
    (0.5)WORDTONUMBER(
      TWO)))

Query "What is the square root of 2 miles in inches add the population of the capital of France times TWO?"
-> ['24134355.9775'] returned in 4.68 seconds
I think the answer is 24134355.9775.
```

Equality

```
> Is the square root of 100 divided by 2 equal to the square root of 25?

NUMBEREQUAL(
  (0.5)DIVISION(
    (1.0)SQUAREROOT(
      100.0),
    2.0),
  (0.5)SQUAREROOT(
    25.0))

Query "Is the square root of 100 divided by 2 equal to the square root of 25?"
-> ['True'] returned in 1.22 seconds
I think so, yes.
```

Deep Recursion

```
> What is 1 add 1 add 1 add 1 add 1 add 1 add 1 add 1?

ADDITION(
  (0.5)ADDITION(
    (0.5)ADDITION(
      (0.5)ADDITION(
        (0.5)ADDITION(
          (0.5)ADDITION(
            (0.5)ADDITION(
              1.0,
              1.0),
            1.0),
          1.0),
        1.0),
      1.0),
    1.0),
  1.0)

Query "What is 1 add 1 add 1 add 1 add 1 add 1 add 1 add 1?"
-> ['8.0'] returned in 0.23 seconds
I think the answer is 8.0.
```

6 Testing

In order to demonstrate the efficacy and limitations of the implemented system, several components and corresponding training examples were constructed. The components[2] developed to test the system are tabulated in Figure 8 below:

Component	Function	Input Domains	Output Domain
ADDITION	Sums two numbers	*numeric*, *numeric*	*numeric*
CAPITALCITY	Returns the capital city of a country	*word*	*word*
CONCATENATE	Concatenates two strings	*wordstring*, *wordstring*	*wordstring*
CONTAINS	Tests if string contains a sub-string	*wordstring*, *wordstring*	*boolean*
CONVERTER	Converts measurements of length	*word*, *word*, *numeric*	*numeric*
DIVISION	Divides two numbers	*numeric*, *numeric*	*numeric*
INTTOWORD	Converts integers to words	*numeric*	*word*
MULTIPLICATION	Multiplies two numbers	*numeric*, *numeric*	*numeric*
NUMBEREQUAL	Tests if two numbers are equal	*numeric*, *numeric*	*boolean*
POPULATION	Returns the population of a country or city	*word*	*numeric*
REVERSESTRING	Reverses a string	*wordstring*	*wordstring*
REVERSEWORD	Reverses a word	*word*	*word*
SQUAREROOT	Calculates the square root of a number	*numeric*	*numeric*
STRINGLENGTH	Returns the length of a string	*wordstring*	*numeric*
SUBTRACTION	Subtracts two numbers	*numeric*, *numeric*	*numeric*
TESTPRIME	Tests is a number is prime	*numeric*	*boolean*
WORDTONUMBER	Converts words to numbers	*word*	*numeric*

Figure 8: **Components used for testing and demonstration**

For each component, a file containing 25 training examples was written. Each file contained 5 examples of 5 differently formatted messages with the same meaning, i.e., 5 differently phrased questions or instructions. A transcript of the training file for the ADDITION component is shown in Figure 9 below:

```
[5:ADDITION]
What is 2 add 6? == 8
What is 1 add 2? == 3
What is 3 add 5? == 8
What is 100 add 200? == 300
What is 15 add 20? == 35
Tell me what 2 add 3 is == 5
Tell me what 1 add 56 is == 57
Tell me what 2 add 4 is == 6
Tell me what 100 add 23 is == 123
Tell me what 2 add 5 is == 7
What is the sum of 2 and 1? == 3
What is the sum of 8 and 9? == 17
What is the sum of 2 and 8? == 10
What is the sum of 80 and 20? == 100
What is the sum of 30 and 40? == 70
Sum 2 and 3 == 5
Sum 10 and 11 == 21
Sum 5 and 6 == 11
Sum 2 and 6 == 8
Sum 20 and 150 == 170
Add 2 and 5 together == 7
Add 8 and 8 together == 16
Add 2 and 3 together == 5
Add 10 and 10 together == 20
Add 23 and 34 together == 57
```

Figure 9: **Training file: *train_addition.txt***

The first line of each agent training file stipulates the agent and the target number of views that agent should acquire as a result of the system training on the examples in the file. When the file is used for testing instead of training, the same agent is expected to provide

[2]The components CAPITALCITY, CONVERTER and POPULATION were programmed with data sufficient for testing only; the components do not make use of large databases and therefore have limited use.

the system with a correct response to each message. It is important to emphasise that the *entire system* is trained and tested on each examples message in each file, not the agent the file corresponds to. The data at the top of each training file is used for testing purposes only; if an example message cannot be correctly answered during testing, the the agent responsible for the failure is known.

Due to the symbolic nature of the design, the set of example messages and corresponding answers used for training were also used for testing. Ideally, two disjoint sets would have been preferable, but after consideration of the the nature of the design, the results of preliminary testing and time available, the decision was made to use only one set of training data. Provided there are more than one training example per message format, or view, this is not thought to pose a significant problem.

6.1 Test Suite

A *Test Suite Class* was added the the Python implementation, see Figure 6.1, to test the system on training files. Each file tested by the suite is scanned for training examples and *links* to other files, which are recursively tested if found. After requesting for a message within a training example to be parsed by the system, the final answer the system gives is compared to the correct answer provided in the training example. If the system returned *only* the correct answer, the system is said to have 'passed' the training example. Each parse is timed and an average parsing time is calculated for each file relating to a specific agent. Additionally, the difference is calculated between the number of actual activated views belonging to an agent and the target number of views in each training file, see Figure 9 on page 24.

After a file linking to several agent training files has finished being tested, a summary is displayed containing details of any failures occurring during the training, and statistics relating to each agent tested: the percentage of messages correctly answered, the average time taken to parse each message and the difference in between actual and target number of views. This data is also output to the file *test_results.txt*, containing comma separated values.

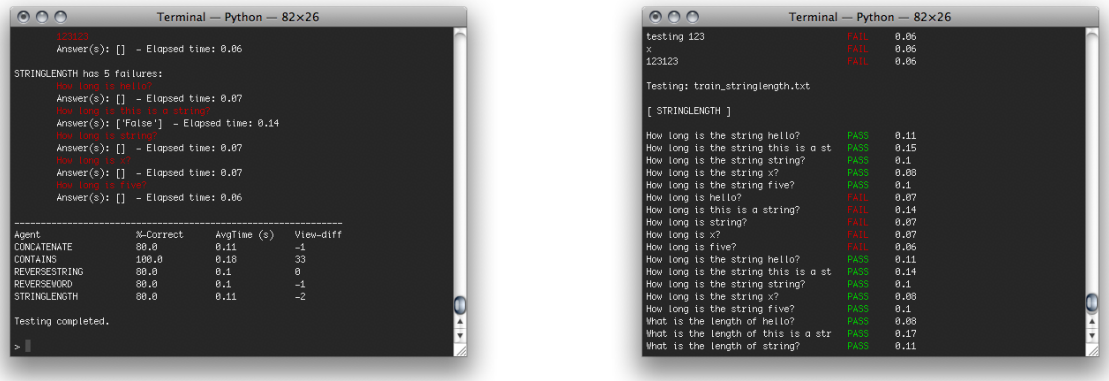

Figure 10: **Screenshots of the Test Suite**

6.2 Test Results

After preliminary testing was carried out to ensure each component was functioning correctly outside the system, the components were installed and trained. The system view activation threshold α was varied from 1 to 5 in order to test training efficacy. The results where alpha was set to 1, 3 and 5, are graphed in Figure 11 below:

Figure 11: Test results for all training examples, where $\alpha = 1, 3, 5$

The results demonstrate that for higher values of α the training was more successful: more training examples were correctly answered, the average parse times were lower and the number of target views were more closely matched. However, even in the optimal case, several anomalies were present. The results where $\alpha = 5$ are tabulated in Figure 12, page 27, and discussed further.

Component	%-correct	Avg Time (s)	Target difference
ADDITION	100.0	0.12	0
DIVISION	100.0	0.1	0
MULTIPLICATION	100.0	0.12	0
SUBTRACTION	100.0	0.1	0
SQUAREROOT	100.0	0.1	0
TESTPRIME	100.0	0.13	0
NUMBEREQUAL	100.0	0.14	0
CONCATENATE	80.0	0.11	-1
CONTAINS	100.0	0.18	33
REVERSESTRING	80.0	0.1	0
REVERSEWORD	80.0	0.1	-1
STRINGLENGTH	80.0	0.11	-2
CAPITALCITY	100.0	0.09	-2
CONVERTER	100.0	0.12	0
POPULATION	100.0	0.11	0
INTTOWORD	80.0	0.1	0
WORDTONUMBER	100.0	0.09	0

Figure 12: **Results, where** $\alpha = 5$

Where less views were acquired than intended, such as with the StringLength and Concatenate components, a corresponding drop in the number of correctly answered training examples is observed. For the Contains component, the results show a large number of superfluous views were formed and activated, though this did not appear to significantly affect the percentage of training examples correctly parsed. As an experiment, the Contains component was removed from the system, which resulted in a large increase in parsing speed; parsing was shown to be approximately 8 times faster.

The components demonstrating less than 100% correct response were investigated and a limitation in the design of the system was discovered. During the verification phase of training, the views that were not present during testing were deleted due to a conflict between two very similar views.

7 Evaluation

7.1 Future Enhancements

In order to make the system robust to spelling mistakes and typing errors, a *fuzzy matching* algorithm could be used to complement or replace the present view matching algorithm. Several prototypes were made and worked well in testing, but at the detriment of the parsing speed. The algorithm was implemented, but not included in the final design due to insufficient time.

Another layer of generalisation *between* views could be taken advantage of, as agents' views often share several similarities. My analysing these similarities, 'synonyms' can be established. For example, many agents can be trained to respond to messages beginning with "What is" and "Tell me what", which are removed during the standard generalisation used in the design. By linking these two synonymous sub-strings together and comparing results with other agents, several synonyms could be collectively established and subsequently used to elaborate agents' existing views.

Due to lack of time available, the algorithm used for view formation was simplified significantly. A more advanced algorithm could be used to intelligently search through compositions involving multiple components in order to form views, where the training data already obtained can be used as a guide to more efficiently search through the composition space.

To enhance the ability of the system further, a decision tree using ID3 or C4.5 could also be used within each agent, or in the system, where a correct composition must be chosen among many possible compositions. Such a decision tree could take into account specific views used, as well as agents, in order to learn exceptions going beyond the equations currently used to determine the correct component.

7.2 Conclusion

An implementation of the proposed design was demonstrated to correctly respond to arbitrary messages requiring the use of multiple functions, after being trained with example messages and corresponding correct answers. Although some irregularities were observed during training, the initial objective was nonetheless achieved.