Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

MSc Thesis

Learning in Dialogue Interactions

Author:
Dario Chiappetta

Supervisor:
Prof. Raquel Fernández

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Institute for Informatics
Institute for Logic, Language and Computation

November 2013

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITEIT VAN AMSTERDAM

Abstract

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Institute for Informatics
Institute for Logic, Language and Computation

Master of Science

Learning in Dialogue Interactions

by Dario Chiappetta

A basic fact about human-human dialogue is that there is often more than one way of talking about any domain. For example, instead of saying *"Turn right after 200 meters"*, a route giver may say *"Turn right at Barclays"* given that *"Barklays"* is an enstabilished referring expression. Human speakers efficiently align terminology and associated ontology in talking about any specific domain (in this example we can say that ontologically there is only one entity – a location – but there are different linguistic terms we can use to refer to it). In contrast, current dialogue systems typically have a static ontology and a static vocabulary, which is used in both generation and interpretation, requiring users to formulate their utterances using the terminology known by the system. The goal of this project is to work towards a system that adapts its own linguistic resources (including possible ontology, vocabulary and grammar) to the interlocutor. The system should be able to learn new concepts by assigning new meanings to known words, as well as new words to talk about concepts known by the system in the domain.

Acknowledgements

The acknowledgements and the people to thank go here…

Contents

List of Figures

Abbreviations

AI	**A**rtificial **I**ntelligence
ASR	**A**utomatic **S**peech **R**ecognition
CA	**C**onversational **A**gent
DM	**D**ialogue **M**anager
DP	**D**ialogue **P**artner
ISU	**I**nformation **S**tate **U**pdate
LU	**L**anguage **U**nit
ML	**M**achine **L**earning
MT	**M**achine **T**ranslation
NLP	**N**atural **L**anguage **P**rocessing
SLU	**S**poken **L**anguage **U**nderstanding
TDM	**T**RINDIKIT **D**ialogue **M**anager
TTS	**T**ext **T**o **S**peech

Symbols

c	chunk of text
m	meaning
s	sentence
σ	chunk similarity
τ	word similarity

Chapter 1

Introduction

The **user interface**, or human-computer interface, is the component of a computer system that provides a space of interaction between the human user and the resources offered by the machine; such a space defines a bridge language which human intentions can be translated into, to be converted into computational procedures for the machine; vice versa, the result of the computation is then presented to the user in the same language, which he or she is assumed to understand.

In the early days of computing, the so-called **batch interfaces** were non-interactive: the user/programmer was supposed to feed the machine with a software, punched on cards using the *machine's assembly language* directly, and retrieve the result of the computation printed on paper. The third and fourth generations of computing brought **text-based** interfaces for operating systems (UNIX, DOS, CP/M), that were later taken over by **Graphical User Interfaces** (GUI), moving the human-machine interaction on a new, visual language made of windows, icons and buttons. Recently, touch screen and camera devices allowed for the implementation of even more natural means of interaction based on visual and **haptic** technologies.

From this brief spot on computing history, and in a way from common sense, we can draw the rather trivial, yet crucial, conclusion that the trend in user interfaces development is to **close the gap** between humans and machines by moving the needle of the interface languages from a machine-centered space towards the human language itself. Due to this, the studies on Natural Language Processing (NLP) assume a dramatically central

```
["dream", 3, [
 ["*", [
     "What does that dream suggest to you ?",
     "Do you dream often ?",
     "What persons appear in your dreams ?",
     "Do you believe that dreams have something to do
      with your problem ?"
  ]]
]]
```

FIGURE 1.1: One of the patterns included in ELIZA's DOCTOR script. It simply tells the system that the user input can be answered with any of the given sentences, whenever it contains the word "dream".

role, as dramatically central is natural language in the interaction of humans with each other.

For this reason, the focus of this thesis work concerns the area of **dialogue systems**.

1.1 Dialogue Systems

Research on dialogue systems has been carried on since the **early days** of Artificial Intelligence. A milestone in the early work on this field is ELIZA (Weizenbaum, 1966), which provides the user with a basic human-like interaction based on pattern matching; Figure 1.1 shows an instance of these patterns.

Another example is the SHRDLU system (Winograd, 1971), which interfaces the user with a simple spatial domain by listening to the user's utterances (e.g. "Would you please put the green pyramid in the box?"), and performing actions accordingly in the domain, resolving, if necessary, ambiguous or implicit references to the entities in it.

According to Jokinen and McTear (2009), modern dialogue systems can be divided in **two main types**: task-oriented and nontask-oriented. Intuitively, systems in the first category are meant to deal with a specific task such as making a hotel booking, or booking a plane ticket; an example in this category is the MIT Mercury system, a vocal interface to a flight database (Seneff and Polifroni, 2000). On the other hand, nontask-oriented systems are meant to engage in conversations without a specific purpose to fulfill, but the one of delivering a realistic simulation; ELIZA itself is an example of nontask-oriented dialogue system.

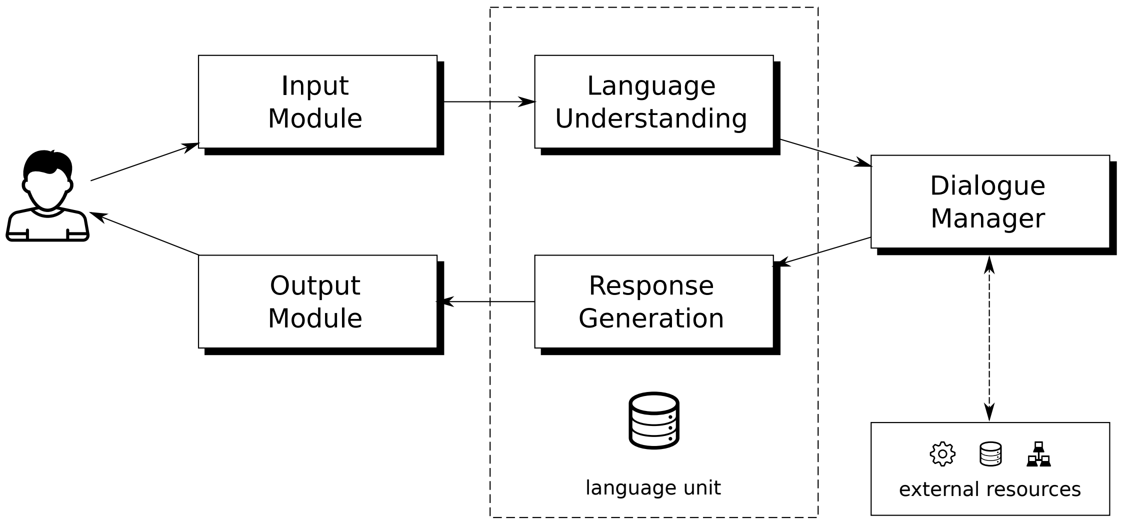

FIGURE 1.2: The basic architecture of a dialogue system

Task-oriented systems can be very simple, as simple and well-formalized the task is; many applications, such as travel service or call routing, can be successfully solved by **slot-based** systems: each step of the conversation requires some pieces of information, modeled as slots, to be filled in by the user (departure city, arrival city, date, and so on); given the slots to be filled, the dialogue task can be solved with a formal grammar of interaction. As the complexity increases, more phenomena of human interaction have to be modeled, such as turn-taking, multimodality or grounding , as well as semantic structures such as quantification and negation; slot-based systems are not sufficient to model these scenarios (Gabsdil, 2003), that require more advanced frameworks such a the Information State Update (ISU) one, which will be tackled in 2.3.1.

1.2 General Architecture of a Dialogue System

A dialogue system, or conversational agent (CA), is a computer software that allows humans and machines to interact through an intermediate language which is as close as possible to the **human language**, and through conversational episodes that implement as close as possible the human dialogue modalities.

The **basic architecture** of a dialogue system is shown in Figure 1.2. The figure represents the flow of information during the interaction between the system and a human user.

1. The user produces an utterance to be entered in the system, that receives it through its **input module**. As an example, the user may start a conversation by asking *"What's your name?"*. Note that, although we will only address text-based interaction, spoken dialogue systems exist, where the input module is an Automatic Speech Recognition software.

2. The **language understanding** module receives the user utterance, and interprets it into a logical form that the system can work with. In the previous example, the language understanding module would receive the string `"What's your name"` as input, and will output a formal representation of the dialogue act performed by the utterance, for instance something like `"ask(?X.my_name(X))"`.

3. The result of the interpretation is given to the **dialogue manager** (DM). This component is the core of the system, as it has to model and keep track of the dialogue context and, within this context, integrate the user moves and plan the system responses. The dialogue manager can also access external resources in order to satisfy the user requests. In the example, the DM receives the interpreted user move `"ask(?X.my_name(X))"`; its will then process this information, which broadly means:

   (a) Update the context, to include the pending user request.

   (b) Find an answer for the user request

   (c) Output the answer in the form of a system move (in the example, `"answer(my_name('HAL9000'))"`)

4. The system move is received by the **response generation** module, whose purpose is to express that move with a natural language sentence, In the example, a proper output would be the string `"My name is HAL9000"`.

5. Finally, the response string is received by the **output module**, which presents it to the user. This module can simply print the sentence on the screen, but it can also implement Text-To-Speech technology to allow spoken interaction with the user.

As we will see in the next chapters, our work will be focused on the language understanding and response generation modules, that together form what we call the **Language**

Unit. Our goal is to implement advanced language understanding and learning capabilities for this part of the system. As a part of the learning capabilities of the system involves interaction with the user (see Section 2.3.2), the dialogue manager was also extended to support such interaction (see Chapter 5).

1.3 Learning to talk

One of the constituent features of humans' ability to speak is that such an ability doesn't come fully developed in children, but rather **grows** with time, influenced by the interaction of the subject with the outer world.

Ever since the power of computers grew enough to allow for intensive statistical analysis of significant amounts of data, **Machine Learning** approaches to Artificial Intelligence tasks got more and more prominent in the scene, often outperforming static methods (i.e. where the solution procedure for a task is explicitly coded by the programmer). One of the clearest examples is the field of NLP itself, where the most important tasks, like parsing or machine translation, are dominated by Machine Learning methods based on corpora, meaning that, for instance, a Machine Translation system will first be trained on a corpus of aligned sentences. Statistical structures will be extracted from this corpus, like the most likely word-by-word alignment, and later be used to process new examples.

However, the task of learning through dialogue interactions comes with some **peculiar** challenges. First of all, humans learning to talk do not go through two separate phases of learning and processing, but rather improve their abilities episode by episode; as Fernández et al. (2011) point out, this **incremental learning** structure is nowadays not implemented in state-of-the-art systems. Also, the nature of this incrementing learning is **not linear**, as the new information was stacked little by little on the existing one: new words and phrases can be described with concepts and linguistic structures that are already present in the learner's mind. Lastly, if we keep considering the way humans learn to talk, we realize that, as from a certain point, the language ceases to be a mere subject of learning, and **becomes the means** by which it is itself learned: the same linguistic structures that are used to express concepts and categories of the learner's experience, can also be used to describe themselves, as they become part of the same experience; we can observe a clear example of this process in any primary-school-level English class, where the teacher explains, by using English sentences, how English sentences are structured.

We can identify some **ideal features** we would like to have implemented into an automatic language learner:

- Learning to produce or understand new surface forms, or **realizations**, for a given meaning – eg. the sentence "Bill eats an apple" for the action of Bill eating an apple.
- Learning to produce new **meanings** from the existing ones and their respective realizations – eg. the concept of motor home, sharing the features of a house and a car.
- Learning a **"grammar" of conversation**, to place the correct utterance at each step of a conversational episode. – eg. an appropriate answer to the utterance "My name is Bill" can be "Nice to meet you", whereas "I like cookies" would not sound as much appropriate.

Lastly, we can point out that when we talk of such things like *realizations*, *meanings* and *episodes* we cannot provide exact, sharply bounded definitions, as it can be argued for **recursive and compositional** structures at any level of their interpretation. As an example, let's consider the sentence

(1) I am pronouncing a sentence that contains two verbs and three nouns

The meaning of such a sentence describes structural elements of the sentence itself (which is the realization of the same meaning) and also the conversational episode that starts when the sentence is stated.

1.4 This thesis project

The aim of this thesis project is to design and implement language learning capabilities for an existing dialogue system, focusing on the **realization level**. That is, given a fixed list of meanings, the system should be able to classify every given sentence into its correct meaning. A client application has also been developed, that makes use of the dialogue system to solve a real user experience task.

Such an application is a voice-controlled music player, which has been named **SVPlay** . The task of such an application is to to get natural language input from the user and translate it into an appropriate corresponding behaviour. For example, when the input is “Play Pictures at an exhibition”, the system should start playing the famous suite by Modest Mussorgsky.

In this domain, each **meaning** corresponds to an action that the player can perform (e.g. play a song, jump to the next track, increase the volume, etc.), and is defined by a set of representative sentences, being its surface forms. For instance, the action of increasing the volume level can be defined by the following set of sentences:

(2) Increase the volume

(3) Increase the volume level

(4) Raise the volume

(5) Increase the volume please

The **task** for the application is, given an arbitrary input sentence and a context (the point of the conversational episode being realized), to reply appropriately, and perform the correct action, that is, **associate** that sentence with its correct meaning. Furthermore, the system should be able to **learn** new realizations for each meaning, as unknown sentences are given in input and processed.

Note that such processing might be more or less **semantically intensive**. As an example, it can be argued that, given the above definition of the action to increase the volume, matching *“Raise the volume please”* is an easier task than classifying *“Turn up the volume”*. This is because the first sentence can be seen as a mere, string-wise, fusion of the two existing examples *“Raise the volume”* and *“Increase the volume please”*, whereas the second one requires a model of *“Turn up”* being a string that carries the same meaning as other strings like *“Increase”* or *“Raise”*.

Also, an unknown input sentence should be given a **confidence score** for each candidate meaning it is associated to, in order for the system to model the uncertainty in the classification of unknown examples. This is important because, depending on the degree of confidence of an interpretation, the system may change its interaction plans (e.g. asking the user for clarification).

Finally, the system should be able to narrow possible needs for **clarification** down to single sentence components, eventually asking the user for disambiguation as specifically as possible. This is to enforce the learning of small components that may appear again in further unknown examples.

1.4.1 Overview

This work is structured as follows. Chapter 2 reviews the **related work** that has been previously done; chapter 3 describes the **architecture** of the solution that has been developed; chapter 4 delves into the **M2 algorithm**, which is the core of the meaning matching feature of the application; chapter 5 describes how the system can learn though the **interaction** with the user; chapter 6 draws the **conclusions** of the whole project and illustrates future work that could take steps from it.

Chapter 2

Related Work

The goal of this thesis, to design and implement a **learning-capable** dialogue system, combines different disciplines within the fields of Artificial Intelligence and Linguistics. This chapter reviews the most relevant work that has been previously done, and that contributed to the realization of this project.

This work includes research on **sentence similarity** measures (2.1), corpus-based approaches to **language processing** (2.2), with a particular attention to the IBM Watson structure, that inspired the meaning matching part of the software, and, of course, literature about the Information State Update (**ISU**) approach to dialogue management (2.3), which constitutes the basic framework of the application we developed.

2.1 Sentence Similarity

As it has been mentioned in the previous chapter, the core task of the system is to associate an unknown sentence with its correct meaning, where each meaning is defined by a set of sentences realizing it. Therefore, one of the constituent capabilities that the system must implement is the ability to tell whether two sentences **share the same meaning** or not.

The problem of scoring the similarity between two sentences is not new in the literature, and a number of different approaches already exist to tackle it. Achananuparp et al. (2008) suggest to classify the existing measures in **three categories**: word overlap measures, TF-IDF measures and Linguistic measures. **Word overlap** scores are computed taking into account only the number of words that are shared between the two input sentences; a basic measure of this kind is the Jaccard coefficient, which is defined as the size of the intersection of the words in the two sentences compared to the size of the union of the words in the two sentences. Banerjee and Pedersen (2003) extended the concept to include a special treatment of phrasal n-word overlaps, motivated by the fact that they are much rarer than single word ones. **TF-IDF** measures are based on term frequency-inverse document frequency, hence the name. Those are common measures to express the importance of a term in a document in an indicized corpus; respectively, they represent the frequency of the term in the document, and the frequency of the term across all documents. TF-IDF can be used to score the similarity between two sentences, for instance, computing the cosine similarity in a vector-space approach. Lastly, **linguistic** measures are meant to exploit, intuitively, the linguistic information contained in the input sentences. Such information consists of semantic relations between words, and the syntactic structure that connects them.

The way sentences are compared in SVPlay takes into account aspects of all these three types of measures, which are combined together in a feature-oriented fashion; the specific algorithm for sentence comparison is described in Chapter 4.

2.2 Machine Learning for Language Processing

The task of labeling an unknown sentence with its correct meaning can be easily expressed in terms of Machine Learning. In fact, it is a standard supervised **classification problem** to learn a class' model from examples, and later use that model to label new data points. In this view, a data point is a natural language sentence, and a label is its meaning.

2.2.1 Parsing and Machine Translation

A great source of inspiration for this work is represented by **statistic**, corpus-based methods in Computational Linguistics; a significant example comes from The IBM models for Statistical **Machine Translation** (Brown et al., 1993), that first introduced the idea of feeding statistically intensive **Machine Learning** algorithms with big data from corpora, which nowadays is the dominant paradigm in MT; insightful is also the work on Data Oriented Parsing, and particularly the U-DOP model for **Unsupervised Language Learning** (Bod, 2006), whose core idea is to initially assume all the possible syntax trees for a set of sentences as equally possible, and then use all the possible sub-trees of them to compute the most probable parse trees, letting the structure of the language emerge from the data.

This work in Computational Linguistics is well reflected in our project, as a strong part of it consists of finding **alignments** of strings sharing similar meanings. We will see (Chapters 3 and 4) that the computation of these alignments happens in a similar setting as in corpus-based Machine Translation: every alignment is initially considered plausible[1], but recurrent ones are reinforced more often, and will thus achieve better and better scores as the model is trained with more examples. Also, the same procedure that computes the alignments is used to let **syntactic structures** emerge from training data, and from processing new input sentences.

2.2.2 IBM Watson

Particularly inspiring for the development of this thesis was the work done by IBM on Watson. **IBM Watson** is a question answering computer system that applies advanced Artificial Intelligence techniques to the field of open domain question answering (Ferrucci, 2011). That is, a software capable of crawling a database of knowledge looking for an answer to any specific English question given as input; along with the answer, the system outputs also a confidence value, that accurately reflects the probability of the answer being correct. Watson was in the headlines in 2011 for competing in the popular American quiz show *Jeopardy!*, defeating former winners Brad Rutter and Ken Jennings, and thus winning the $1 million first prize.

What is interesting about Watson, in connection to our work, is the feature-based approach to **evidence scoring** that is used to select the correct answers. Figure 2.1 (Ferrucci et al., 2010) shows a bird-eye view of the whole Watson architecture; the part of this process that is especially interesting for us is the *Hypothesis and evidence scoring*: at that point, the system receives a set of candidate answers for the input question; each of this answers will be run through a series of procedures whose purpose is to find

[1]It is worth to note that, where in classic Machine Translation every alignment is initially considered possible with equal probability, here we use some heuristics to provide a clever initialization, in order to cope with the limited amount of training data we have.

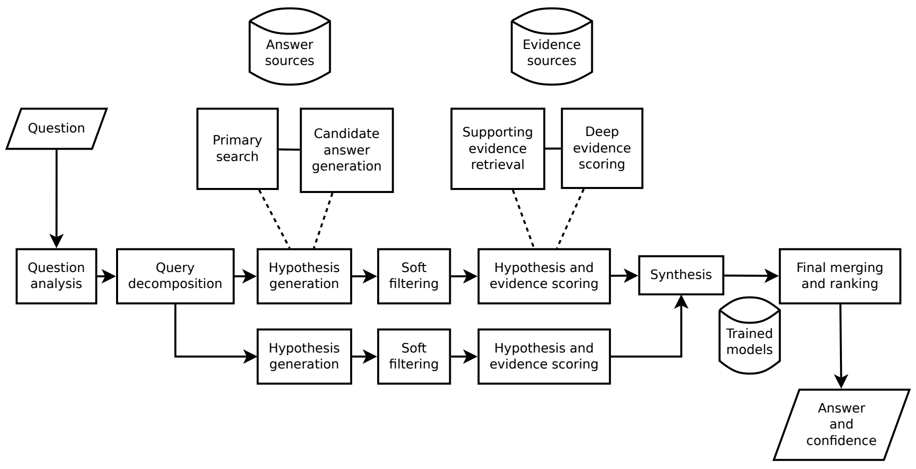

FIGURE 2.1: Watson's high level architecture

evidence supporting that answer. Most of these procedures are not particularly sophisticate, in fact the clever part of the algorithm is to combine a **high number of features** to obtain an accurate final score value. This is done by training Watson's hypermodel with data from previous *Jeopardy!* games; the hypermodel consists of a set of weights, that are then used to produce the optimal linear combination of the features.

Even though this latter meta-learning aspect is not implemented in our work, we will find a similar feature-based scoring approach in the core matching algorithm of the Language Unit, discussed in detail in Chapter 4.

2.3 Dialogue Interaction

As we have mentioned in Section 1.1, Dialogue Systems have a long history in the Artificial Intelligence literature. Section 2.3.1 will introduce the Information State Update framework for dialogue management, on which our software is based. Section 2.3.2 will present some work on grounding and clarification, which was useful to develop the clarification procedures of SVPlay.

2.3.1 Information State Update Framework

The ISU approach to Dialogue Management, as it is described by Traum and Larsson (2003), can be seen as an attempt to reduce the gap between "theories of dialogue that linguists or philosophers of language may devise and theories directly implemented in dialogue systems".

The linguistic and philosophical roots of this theory can be found in the notion of **common ground**, as it is defined by [STALNAKER CITE]; on Stalnaker's account, the common ground consists of an unstrucrured set of all the possible worlds that are compatible with the propositions asserted so far in the dialogue. A slightly more sophisticated formulation of the same concept comes from [LEWIS, CITE], who, drawing a parallel between dialogues and baseball, introduces the notion of "conversational scoreboard", to keep track of the participants' moves. Even closer to the model of Traum and Larsson is **Ginzburg's dialogue gameboard** (Ginzburg, 2012); one of the main differences between this model and the other two is represented by the inner structure of the scoreboard, which in Ginzburg is not just a set of propositions, but rather provides a structure made of facts, questions under discussion (QUD) and moves.

Traum and Larsson's notion of **Information State** (IS) takes Ginzburg's DGB one step further, by extending it with a formal representation of what Ginzburg calls the unpublished part of a dialogue partner's (DP) mental state (UNPUB-MS), and that will become the private part of the Information State. Figure 2.2 shows a simple Information State type, as it is implemented in the IBiS1 system (Larsson et al., 2002, p. 36). It can be easily noticed that the IS is split in two sections, "Private" and "Shared", each of them containing formal representations of informational components: an agenda of actions being executed, a plan for future actions and a set of beliefs for the **private** part, the set of propositions in the common ground, a stack of questions under discussion, and the last utterance in the **shared** part.

Besides the notion of IS, the ISU framework includes two **additional components**: a set of dialogue moves and a set of update rules.

The central concept of ISU-based systems is to **update** this, initially empty, Information State, according to the dialogue moves performed by the DPs. A **dialogue move** is tipycally the formal interpretation of a natural language utterance (e.g. the English sentence "What's your name?" can be interpreted as the dialogue move "`ask(?X.my_name(X))`"[2]), but, in more complex systems, can be also derived from a different type of interaction

[2]Let's not enter the detail of the λ-calculus-like representation of the user move: what is important is that it is expressed in a form that the system can work with.

$$\begin{bmatrix} \textsc{private} & : & \begin{bmatrix} \textsc{agenda} & : & \text{Stack(Action)} \\ \textsc{plan} & : & \text{Stack(Action)} \\ \textsc{bel} & : & \text{Set(Prop)} \end{bmatrix} \\ \textsc{shared} & : & \begin{bmatrix} \textsc{com} & : & \text{Set(Prop)} \\ \textsc{qud} & : & \text{Stack(Question)} \\ \textsc{lu} & : & \begin{bmatrix} \textsc{speaker} & : & \text{Participant} \\ \textsc{move} & : & \text{Move} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

FIGURE 2.2: Information State, as it is implemented in IBiS1

(e.g. a gesture). The system receives this moves, and is able to modify the IS accordingly (e.g. inserting a new question under discussion, or marking an action as completed, and so on), by running them through a set of **update rules**, and an **update strategy**, to decide on which rules to apply when more than one can be selected.

This work is relevant to us, as our software builds on the TDM dialogue management library (see 3.2), which implements the Information State Update framework. Our contribution is located at the **interpretation** level, that is, the part which is resposible of receiving the user utterance and converting into a formally defined dialogue move. Particularly, we will extend the existing library with a new interpretation module, that is able to perform fuzzy matching of natural language sentences, and implements an **interactive learning** procedure to exploit the information contained in unknown examples. Once the sentences have been interpreted, standard dialogue moves are returned, to be used in the system as it would happen with the previous version of the interpretation module.

2.3.2 Grounding and clarification

Herbert Clark in Clark (1996) points out how, in the nature of human dialogue interaction, a major role is played by a social component, the activities that, alongside the individual communicative intentions, ensure the coordination between dialogue partners. In particular, **grounding** is the act of enstablishing mutual knowledge among the participants. Gabsdil (2003) points out how this process, while being almost effortless for human speakers, represents a challenge in computer dialogue systems, and argues for the importance of having as natural grounding interactions as possible.

The following example, taken from Gabsdil's paper, shows a grounding act, in the form of a **clarification question** by one of the DPs:

```
A: You turn the second road on your left hand side
B: You mean Marchmont Road?
```

B wants to make sure that both A and himself are talking about the same road: this necessity for clarification may be due to **different reasons**, for instance A might have provided an inaccurate description of his intention, or B might haven't heard correctly A's utterance. Clark identifies **four levels** speaker and listener have to be coordinated on in dialogue: level 1, where participants **enstablish** communication by respectively executing and attending to the communicative behaviour; level 2, where A presents a **signal** to B, and B identifies it; level 3, where B recognises the **meaning** of A's signal; level 4, where A proposes a **joint project** to B, and B considers it.

Our software will be especially concerned with Clark's **third level**, as the user signal, given in the form of written utterances, is assumed to be considered and recognised correctly by the system[3] [4]. It is worth to point out that the software's pragmatic interpretation of the user utterances has a 1:1 correlation with their semantic representation (e.g. the sentence "Increase the volume" is labeled as "`request(increase_volume)`", a label that direclty expresses the intended meaning of the user, that is, assigning to the system the task to increase the volume), therefore levels 3 and 4 de facto collapse into a single one.

One more notion that has been used in our work is Gabsdil's classification of different types of **non-understanding** into cases where no interpretation is available, cases of uncertain interpretation (one interpretation is still available), and cases of ambiguous interpretation (two or more alternatives are available). We will draw on the same classification in 5.1, where we will describe how the software deals with utterances that have not been understood. In particular, we will see that the "no interepretation" case will

[3]The actual implementation of our software allows the replacement of the text input module with a Speech Recognition one; in this case, uncertainty at level 2 is introduced. This case however is not covered in this thesis.

[4]Yet the occurrence of typographical errors is possible. This case can also be located at meaning level, as the signal is still conveyed faithfully from the user to the system.

be answered with a rephrase request, whereas the other two will be solved with a single polar question[5].

Lastly, Gabsdil also argues for **partial clarification questions**, being clarification request that are targeted to an especially misunderstood part of the user utterance. As an example, in the context of music player application, let's consider the dialogue:

```
U: Put on my favourite from Zappa
S: I don't understand, can you repeat?
```

In this case the system's clarification question is generic, and doesn't sound as natural as the one in the following example:

```
U: Put on my favourite from Zappa
S: You mean 'Sheik Yerbouti'?
```

Gabsdil points out how a particular effort is required to implement this type of questions. We will address partial clarification questions in Chapter 5.

2.4 Summary

In this chapter we have reviewed some of the most relevant work that inspired our project.

We looked at **sentence similarity** measures, they are central for our application, as its core meaning matching features relies on the comparison and scoring of the input sentences against existing examples. According to the literature, we perform such comparison taking into account word overlapping, term frequency and linguistic features.

Great inspiration also came from the field of **machine learning**, and particularly from its applications in computational linguistics. We find the same data-driven approach, that is, letting structures and alignments emerge from examples, in statistic machine translation and parsing. Especially relevant has been the work of IBM on Watson, because of its feature-based scoring approach, that inspired ours.

[5]Note that, as Gabsdil also says, the most appropriate way to answer a case of ambiguous interpretation would be with a wh-question; as it will be explained better in Chapter 5, this has not been done mainly for technical reasons.

Finally, research in the field of **dialogue systems** has to be mentioned, as the software we developed is a dialogue system. In particular, we worked with TDM, a dialogue management library that implements the Information State Update framework. We extended this library with grounding and clarification capabilities (other than with the meaning matching features), that are well present in the literature, both from a philosophical/linguistic perspective and from a computational one.

Chapter 3

Architecture

The outcome of this thesis project is SVPlay , a music player application that accepts English utterances as commands, adapts to utterances it has never been exposed to, and learns from them by possibly interacting with the user, thus expanding its initial knowledge of the language. The application has been written in Python 2.7, and consists of a client application for the existing OpenTDM dialogue management library. Such a library supports basic dialogue management based on the Information State Update approach, but has no support for grounding or flexible understanding of unknown sentences. Therefore, TDM has been extended with the Language Unit module, that introduces these capabilities.

Section 3.1 describes the SVPlay client, Section 3.2 describes the OpenTDM library, and Section 3.3 describes the Language Unit module.

3.1 The Client Application

SVPlay is a client application for OpenTDM, a dialogue management library based on the work by Larsson et al. (2002). An OpenTDM application can be seen as a container for domain-specific **parameters** for the dialogue manager. These parameters consist of:

- A **device** class, containing variables and methods that directly control the actions of the application. The *device* file of a music player application will contain, for

```
{
  "goal": "increase_volume",
  "plan": [
              "findout(?X.volume_to_increase(X))",
              "dev_perform(
                           IncreaseVolume,
                           MplayDevice,
                           {postconfirm=True}
                          )"
           ],
  "postconds": ["done(IncreaseVolume)"]
}
```

FIGURE 3.1: The `increase_volume` plan of SVPlay .

instance, variables holding the current playlist, or the current volume level, and methods to play/stop the music, lookup for a song and so forth.

- An **ontology**, whose main purpose is to define predicates and actions that will be used at dialogue management level. In the case of SVPlay , an example of predicate is `current_song(X)`, that identifies the song currently being played (note that such a predicate must be mirrored in the device file as a variable[1]); an example of action is `increase_volume` that, intuitively, identifies the action of increasing the volume (actions are mirrored as well in the device class, in the same way predicates are).

- A **domain** file, whose main purpose is to contain the list of plans that will control the dialogue episodes. For instance, the TOP plan of an application is to find out what the user wants to do. Another example of a plan in SVPlay is the `increase_volume` plan: step 1 of the plan is to ask the user for how much the volume should be increased, step 2 is to perform the actual action through the device.

  Figure 3.1 shows the plan item, as it is defined in the application domain: we notice that the `plan` section is a list of two elements, a `findout()` function, that will ask the user for the amount of volume to increase, and a `dev_perform()` function, that will fire the volume increasing method in the device.

[1]The actual implementation consists of an inner `current_song` class of the device class, which access the proper, private, variable of the device file through a `perform()` method. The reasons that led to such an implementative choice were not made known by the authors of OpenTDM.

- A **grammar** implemented using the Grammatical Framework[2] (Ranta, 2004). This part will not be discussed, since, as it is explained in Section 3.3, the Grammatical Framework in OpenTDM has been replaced with a specifically created module called Language Unit (LU). However, the purpose of the two modules is the same, and covers what in Section 1.2 we called language understanding and response generation. That is, mapping natural language user utterances to their formal representations and vice versa.

  In the example of Figure 3.1, a language understanding use of the Language Unit is to take, for instance, the input sentence "Increase the volume" and turn it into a `request(increase_volume)` user move, that will be used to load the proper plan from the domain. A response generation use of the LU will be, for instance, to take the `ask(?X.volume_to_increase(X))` system move (that will be produced by the `findout()` part of the plan), and turn into a sentence like "By how much you want the volume to be increased?".

The dialogue management logic is left to OpenTDM, that will be discussed in the next section.

3.2 OpenTDM

OpenTDM is a **dialogue management library** developed and maintained by Talkmatic[3] based on the Information State Update framework (see 2.3.1). OpenTDM is a Python 2.7 port of the TRINDI Dialogue Move Engine Toolkit (Larsson and Traum, 2000), that was originally implemented in Prolog.

The purpose of OpenTDM is to keep track of the Information State (IS), updating it on new user utterances, and planning the system moves accordingly. This is done by applying the standard ISU rules and policies (Larsson et al., 2002) to the resources defined by the application (ontology, domain, device and language). Appendix A presents an example of how SVPlay's information state changes during an interaction with the user.

[2]http://www.grammaticalframework.org/
[3]http://talkamatic.se/

It is worth to note that OpenTDM is not yet available to the public, and comes with **no documentation**. This has significantly limited us in the design of the solution, preventing us from both implementing elaborated extensions of it (e.g. more complex learning interactions than the ones described in Chapter 5), and respecting the software architecture best practicses, as many of the OpenTDM extensions that we have implemented rely consistently on hacks and temporary workarounds.

3.3 The Language Unit

The Language Unit (LU) is a Python library that has been specifically created to support SVPlay 's utterances understanding ability, serving as a drop-in replacement for the Grammatical Framework, that is used in the main OpenTDM branch. The purpose of this library is to perform the classification task as it has been defined so far, that is, assigning every input sentence to its correct meaning label.

3.3.1 Language

The `Language` class is the main class of LU, its purpose is to model a natural language under the following **abstraction**: a Language is defined by a finite set of Meanings (labels); a Meaning is defined by a finite set of Sentences expressing that Meaning. As an example, the SVPlay language consists of a number of meanings, each of them defining an action for the application to perform; one of these meanings can be the one to pause the current song; this meaning will be realized by a number of English sentences, like “Pause the song”, “Suspend the music”, or “Pause the current track”. For the purpose of language understanding, a Sentence can be brougth down to any of its (linguistic or non-linguistic) constituents called Chunks or Phrases. A chunk is formed by one or more Words.

The following are the main capabilities implemented in the Language class:

- **Load** and **save** languages. Each OpenTDM application must define a `language/` folder where the language is stored, in the form of a `.l` file. Such a file is just a dump of all the meanings and their example sentences. Since the language can evolve through learning, applications’ language files are updated every time the application is run and dialogue interactions are performed. This update consists of adding new examples to meanings (unknown sentences from the user that have been understood with a good degree of confidence) and updating the usage count of the known examples that have been processed during the interaction with the user.

- **Learn** a sentence. When a new labeled example (a sentence with its meaning label) is provided, the knowledge of the language is extended. This is done adding the new sentence to the list of sentences realizing that meaning, and drawing statistics (e.g. the frequency of a certain word/phrase in the given meaning) to improve the model of the meaning. Note that, when a language is loaded for the first time, every sentence in it is run through the learning procedure to initialize the statistics.
- **Understand** an input sentence. The core task of the Language Unit is to associate an input sentence with its correct meaning. While this operation is trivial when the input sentence is already present in the language, it becomes hard for unknown examples. In this latter case the LU computes a score for the sentence against each of the meanings that are present in the language; the meaning that achieves the best score is given as an output, along with the score itself, representing the degree of confidence for the output to be correct. The way sentences are scored is presented in the next section.

3.3.2 Scores

The way the Language Unit understands a sentence is based on a number of nested comparisons, resulting into as many comparison scores. A **score** is a linear combination of separate features. The top-level task is to score the sentence against every meaning in the application database; for each of them a **Meaning Score** (see 3.3.2.2) is computed, representing how likely the input sentence is to be a realization of that meaning. This score is based on the similarity of the input sentence with the known sentences realizing the meaning; the **Sentence Score** (see 3.3.2.3) represents the similarity between two sentences. This score is based on how similar the (linguistic or non-linguistic) components of the two sentences are; any sub-string of a sentence is called Chunk, and the **Chunk Score** (see 3.3.2.4) represents the similarity between two chunks. The Chunk

Score is based on the similarity of the words contained in the two chunks; the **Word Score** (see 3.3.2.5) represents the similarity between two words.

3.3.2.1 The `Score` Data Structure

Scores in the application are modeled by the `Score` **class** (`lu/score/__init__.py`). A `Score` object defines the following **fields**:

- `Score.features`, an array of N features that are used to form the final score. A feature is represented as a floating-point (`float`) number.
- `Score.weights`, an array of N weights that are used in the linear combination of features to form the final score. A weight is also represented as a floating-point (`float`) number. Weights are equally distributed by default, but they can be overridden with custom values. To this respect, an ideal scenario would be the one in which **optimal weights** for the features are determined with machine learning (*meta-model learning*), as it will be described in Chapter 6.

Implementation-wise, for the sake of efficiency, the Python `array` module[4] is used instead of traditional lists and dictionaries. Such a module defines "an object type which can compactly represent an array of basic values", directly based on their original C implementation. The acces to the elements of an `array` is positional. For this reason any instantiation of `Score` is recommended to define proper **constants** to access the single features.

The `Score` class also defines the following **methods**:

- `Score.set_feature(i, value)`, sets `value` as the value of the the i-th feature.
- `Score.get_feature(i)`, returns the value of the the i-th feature.
- `Score.get_score()`, returns the final score, as the weighted average of the features:

  $$\sigma = \sum_{i=1}^{N} w_i f_i$$

  Where σ is the final score, N is the total number of features, f_i is the i-th feature and w_i is the weight of the i-th feature.

[4]http://docs.python.org/2/library/array.html

Parallel to its class definition, any subclass of score must define a list $h = \{h_1, ..., h_N\}$ of procedures, called **hooks**, that compute the single features of the score, and a **factory** method, which runs the two elements to be compared through the feature hooks, fills a score object with the results and returns it. As an example, let's consider a fictional Word Score class, that scores the similarity of two English words, and that has only one feature, being the edit distance between the two words. In this case the hooks list will contain only one element ($h = \{h_1\}$), which is a pointer to a procedure `c_edit_distance(word_from, word_to)`, that returns the edit distance between two input words. The factory function `get_score(word_from, word_to)`[5] will instanciate an empty score object, and will assign the result of the computation of every feature hook h_i to the corresponding feature f_i; therefore, the only feature of the score, representing the edit distance, will be filled with the result of `c_edit_distance`.

The following sections describe how the `Score` class is **extended** to model the different levels of scores that were introduced above.

3.3.2.2 Meaning Score

A Meaning Score $\sigma_M(s, m)$ represents the likelihood of the sentence s to be a realization of the meaning m. It is represented in the application by the `MeaningScore` class (`lu/score/meaning.py`), which is a subclass of `Score`. Such a class defines the following **three features**:

- `MeaningScore.MAX_SSCORE` - The best similarity score between the input sentence and one of the sentences that realize the meaning.

  $$\sigma_{M,\text{MAX\_SCORE}}(s, m) = \max \sigma_S(s, s_i)$$

  Where $\sigma_{M,\text{MAX\_SCORE}}$ is the value of the feature, s is the input sentence, m is the meaning s is scored against, σ_S is a Sentence Score (see Section 3.3.2.3) and $s_i \in m$ is one of the know sentences that realize m.

[5]Not to be confused with the `Score.get_score() method.`

- `MeaningScore.AVG_SSCORE` - The average similarity score between the input sentence and the sentences that realize the meaning.

$$\sigma_{M,\text{AVG\_SCORE}}(s, m) = \frac{\sum_{i=1}^{N} \sigma_S(s, s_i)}{N}$$

  Where N is the number of known sentences that realize m.

- `MeaningScore.ML_CCSUM` - The sum of the class-conditional probabilities of every possible sub-string of s.

$$\sigma_{M,\text{ML\_CCSUM}}(s, m) = \sum_{c_s \in s} p(c_s|m)$$

  Where $c_s \in s$ is a sub-string (chunk) of s and $p(c_s|m)$ is the class-conditional probability of c_s respect to the meaning m, that is, the relative frequency of c_s in the known sentences that realize m.

3.3.2.3 Sentence Score

A Sentence Score $\sigma_S(s_{\text{from}}, s_{\text{to}})$ represents how likely sentences s_{from} and s_{to} are to realize the same meaning. Even though more features could be considered to measure this value, at the moment a Sentence Score is defined as the Chunk Score (see Section 3.3.2.4) $\sigma_C(c_1, c_2)$, where c_1 is the chunk spanning the whole s_{from}, and c_2 is the chunk spanning the whole s_{to}.

3.3.2.4 Chunk Score

A Chunk Score $\sigma_C(c_{\text{from}}, c_{\text{to}})$ represents how similar the chunk c_{from} is to the chunk c_{to}. In the context of this thesis, a Chunk is defined as any possible substring of a sentence s (not necessarily syntactic constituents), including s itself. As an example, the sentence

(6) “Execute order 66”

contains the chunks “Execute”, “order”, “66”, “Execute order”, “order 66” and “Execute order 66”.

In the application, a Chunk Score is represented by the `ChunkScore` class (`lu/score/chunk.py`), which is a subclass of `Score`. Such a class defines the following **five features**:

- `ChunkScore.AAVG` - The maximum, over all the possible 2-splits of the input chunks, of the arithmetic averages of the best alignment scores.

$$\sigma_{C,\text{AAVG}}(c,d) = \max_{i,j}\left\{\frac{\max\left\{\sigma_C(c_0^i, d_0^j), \sigma_C(c_0^i, d_{j+1}^M)\right\} + \max\left\{\sigma_C(c_{i+1}^N, d_0^j), \sigma_C(c_{i+1}^i, d_{j+1}^M)\right\}}{2}\right\}$$

  Where c is a chunk of N words, d is a chunk of M words, c_a^b is a sub-string of c spanning from the a-th word of c to the b-th word of c, $i \in (0, N]$ and $j \in (0, M]$.

  By **2-split** of a chunk we mean a split of it in two sub-strings of arbitrary size; as an example, a 2-split of the chunk c “Increase the volume” can be

  (7) | Increase | the volume |

  In this case we refer to “Increase” as c_1^1 (the sub-string of c spanning from position 1 to position 1), and to “the volume” as c_2^3.

  Given two chunks that have been split in two parts each, each sub-string of one can be associated with a sub-string of the other[6]. For example, given the chunks c “increase the volume” and d “turn up the volume”, an ideal behaviour would be to associate c_1^1 (“increase”) with d_1^2 (“turn up”) and c_2^3 (“the volume”) with d_3^4 (“the volume”). Such an association is what we call an **alignment**[7], and it produces two Chunk Scores (in the case of the example, $\sigma_C(c_1^1, d_1^2)$ and $\sigma_C(c_2^3, d_3^4)$).

  Thus, for every possible combination of 2-splits of the two input sentences, a number of alignments is possible, each of them associated with two Chunk Scores. For each combination of 2-splits the alignment that produces the best scores is selected; once this has been determined, the external maximization is performed, so that the best combination of 2-splits is returned.

  It is worth to note that the value of this feature **builds recursively** on the scores of these smaller chunks, chosing the split and the alignment for the final score to

[6]Since a sub-string can be a single word, as well as the whole chunk, we have two special cases to consider. Firstly, a word can be scored against another word; in this case the returned score is not a Chunk Score, but rather a Word Score (see 3.3.2.5). Then, it may happen that the whole first chunk is scored against the whole second chunk; this is prevented by the algorithm, as it would result in an infinite loop.

[7]Note that, while ideally every c_i^j could be aligned with 0, 1 or 2 d_k^l and vice versa, due to implementation restrictions the software allows a c_i^j to be aligned with only one d_k^l, where every d_k^l can be aligned to 0, 1 or 2 c_i^j.

be maximized. For this purpose a special algorithm called M2 has been defined, a more detailed description of it can be found in Chapter 4.

- `ChunkScore.LEN` - Sets a preference for big chunks over short ones.

  $$\sigma_{C,\mathrm{LEN}}(c,d) = 1 - \frac{1}{\min(|c|,|d|)+\alpha}$$

  Where $|c|$ is the length in words of c, $|d|$ is the length in words of d, and α is a smoothing constant.

- `ChunkScore.STRAIGHT` - Sets a preference for straight alignments over crossed ones.

  $$\sigma_{C,\mathrm{STRAIGHT}}(c,d) = \begin{cases} 1, & \text{if } n \text{ streight alignment} \\ 0.5, & \text{if } n \text{ half-straight alignment} \\ 0, & \text{if } n \text{ crossed alignment} \end{cases}$$

  Where, under the assumption that the best alignment has been determined by `ChunkScore.AAVG`, a **straight** alignment happens when the first sub-string of c is aligned to the first sub-string of d, and the second sub-string of c is aligned with the second sub-string of d; a **half-straight** alignment happens when both the sub-strings of c are aligned to the same sub-string of d; a **crossed** alignment happens when the first sub-string of c is aligned to the second sub-string of d, and the second sub-string of c is aligned with the first sub-string of d.

- `ChunkScore.ML_CFREQ` - The Chunk Likelihood. This features is meant to enforce good linguistic splitting of the input chunks, as it sets a preference for chunks that occur more in the language base (eg. the chunk "the volume" is supposed to occur more often that "increase the"). Such a value is based on Machine Learning from previous examples, and it is implemented as the sum of the relative counts of the two chunks. A more detailed explanation of this will be given in 4.2.1.

  Note that this feature does not measure the goodness of the alignment, but should rather be considered as a **language modeling** feature; this has the drawback of contributing to the overall score every times the two input chunks are linguistically plausible, even if they should not be aligned together. Furthermore, a good amount of training data is required to prevent **data sparsity** problems. Both of these issues will be addressed in 4.2.1 as well.

- `ChunkScore.ML_AFREQ` - The Alignment Likelihood. Such a value is based on Machine Learning from previous examples, and it is implemented as the relative count of the times the two input chunks have been found aligned together. A more detailed explanation of this feature will be given in 4.2.3.

3.3.2.5 Word Score

A Word Score $\sigma_W(w_{\text{from}}, w_{\text{to}})$ represents how similar the word w_{from} is to the word w_{to}. It is represented in the application by the `WordScore` class (`lu/score/word.py`), which is a subclass of `Score`. Such a class defines the following **five features**:

- `WordScore.EQUALS` - Boolean equality.

$$\sigma_{W,\text{EQUALS}}(w_{\text{from}}, w_{\text{to}}) = \begin{cases} 1, & \text{if } w_{\text{from}} = w_{\text{to}} \\ 0, & \text{otherwise} \end{cases}$$

- `WordScore.EDIT_DISTANCE` - The edit distance is a measure of string-wise similarity defined by Levenshtein (1966): the more two strings are similar, the less their distance.

$$\sigma_{W,\text{EDIT\_DISTANCE}}(w_{\text{from}}, w_{\text{to}}) = 1 - f(w_{\text{from}}, w_{\text{to}})$$

  Where $f(w_{\text{from}}, w_{\text{to}})$ is the edit distance between w_{from} and w_{to}, as it is implemented in the Python Natural Language ToolKit (Loper and Bird, 2002).

- `WordScore.POSITION_DISTANCE` - inversely proportional to the difference between the position of w_{from} in its sentence and the position of w_{to} in its sentence.

$$\sigma_{W,\text{POSITION\_DISTANCE}}(w_{\text{from}}, w_{\text{to}}) = \left(\frac{1}{|p(w_{\text{from}}) - p(w_{\text{to}})|}\right)^{\alpha}$$

  Where $p(w)$ is the position of the word w in its sentence, and α is a free parameter.

- `WordScore.WN_MAX_PATH_SIMILARITY` - The length of the shortest path between the two words in the WordNet onthology (Miller, 1995). This measure is implemented in the NLTK as well.

- `WordScore.ML_AFREQ` - The Alignment Likelihood. This feature has already been described in 3.3.2.4.

3.4 Summary

In this chapter we described the architecture of SVPlay in general, and of its sentence scoring system in particular.

SVPlay is a client application for the **OpenTDM** library. The application defines its plans, language and device actions, while OpenTDM takes care of the dialogue management, being the execution of the plans that are appropriate to the context (e.g. to increase the volume when the user issues “Increase the volume”).

The **Language Unit** is a drop-in replacement for OpenTDM’s default language processing system (the Grammatical Framework); its main duty is to provide flexible understanding of user utterances. This is done through nested, score-centric, feature-based comparisons between the input sentence and the sentences representing each of the meanings available in the application. The interpretation of an input sentence is given as a list of Meaning Scores, representing how likely the sentence is to realize each of the meanings.

Figure 3.2 shows a breakdown of a **Meaning Score**. From this figure we can see that a Meaning Score is made of three features, one of which relies on Machine Learning data (`ML_CCSUM`), and the other two (`MAX_SSCORE` and `AVG_SSCORE`) are based on single Sentence Scores, representing the similarity between the input sentence and the ones in the meaning. A **Sentence Score** is just a Chunk Score. A **Chunk Score** represents the similarity between two chunks of text, and has five features; two of them make use of ML data (`ML_CFREQ` and `ML_AFREQ`), and one is derived either from other, smaller, Chunk Scores, or from a Word Score (the way smaller scores are combined into a bigger one will be better explained in the next chapter). A **Word Score** has four features, one of which (`ML_AFREQ`) makes use of ML data.

The result of every comparison is also used to update the ML model (e.g. incrementing the count of the chunks that have been encountered). This aspect will be tackled in 4.2.

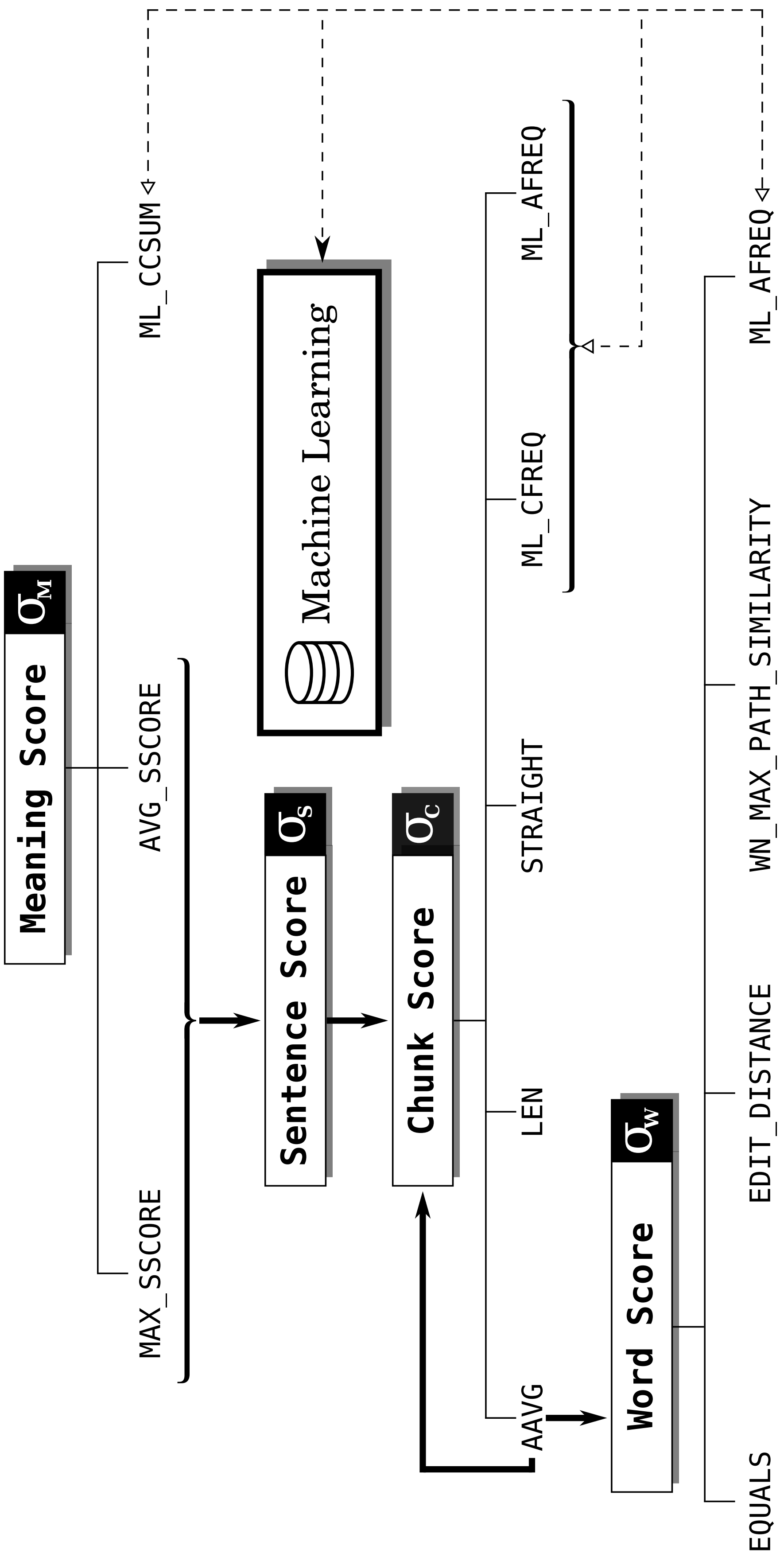

Figure 3.2: Scores' structure

Chapter 4

Sentence similarity computation

In the context of sentences **classification**, where the task is to label an unknown input sentence with the correct meaning, and each meaning is defined by a set of representative natural language sentences (as it was defined in the Introduction), one crucial point is to measure the similarity of the input sentence with each of the sentences defining each meaning.

Computing a **similarity score** between two natural language **sentences** is not a trivial problem. Many features can be taken into account to solve it, such as the syntactic structure of the sentence, or the meaning of the single words, or their order (see 2.1); all of these features are informative respect to the meaning of the sentence, and it is reasonable to assume they are all considered by humans solving the same task. Here we describe a recursive, dynamic programming **algorithm** for sentence comparison aimed to exploit word-based information, as well as the syntactic one, expressed in the form of unsupervised learned binary trees.

The algorithm presented in this thesis aims to provide a similarity measure for two sentences s_1 and s_2 featuring many of the ideas that have proven to be successful in recent developments of Computational Linguistics. The following are some of the key insights of the algorithm:

- **Every possible** sub-sentence of s_1 is compared with every possible sub-sentence of s_2.

- A score between two chunks of text is build **recursively** on the best matches of smaller chunks, and always brought down to scores between single words.
- **Dynamic programming** is used to increase the efficiency of the algorithm, preventing it from computing the same result two times.
- Other than the similarity score, the algorithm outputs the most probable chunking of each sentence in the form of an unlabeled binary **parse tree**.
- Other than the similarity score and the parse trees, the algorithm outputs the most likely **alignment** between the chunks of the two sentences.
- Every score is expressed as a linear combination of **features**, designed to be independent and extensible.
- The partial computations of the algorithm can be stored and further used to implement a learning model.

This chapter is **structured** as follows. Section 4.1 describes the algorithm theoretically. Section 4.2 describes possible extensions and improvements. Section 4.3 draws the conclusion of the paper.

4.1 The M2 Matching Algorithm

Algorithm 1 contains the main loop of the sentence scoring algorithm. Before entering the details of it, it is necessary to say that:

- The input c_1, c_2 can be any **chunks** of text because, while the algorithm is called with two sentences in input, it will recursively call itself on every possible chunk extracted from the two sentences.
- The input T (T for Table) is a data structure holding the intermediate results of the algorithm. As we can infer from line 12 of Algorithm 1, $T[c_1, c_2] = \sigma(c_1, c_2)$
- $\tau(c_1, c_2)$ returns a Word Score, being a similarity measure between two words (see Section 3.3.2.5). As we have seen in Section 3.3.2, every Score is a linear combination of independent features. The features that are currently defined for Word Scores are described in Section 3.3.2.5.

Algorithm 1: The main algorithm

```
input  : $c_1, c_2$ chunks of text; $T$ partial results table
output: $\sigma(c_1, c_2)$ Similarity score between $c_1$ e $c_2$
1  if c1,c2 are words then
2      return $\tau(c_1, c_2)$
3  $S \leftarrow [\,]$;
4  for $i \leftarrow 1$ to Length($c_1$) do
5      $C_1 \leftarrow$ Split($c_1$,$i$);
6      for $j \leftarrow 1$ to Length($c_2$) do
7          if $i$=Length($c_1$) and $j$=Length($c_2$) then break;
8          $C_2 \leftarrow$ Split($c_2$,$j$);
9          foreach $sc_1$ in $C_1$ do
10             foreach $sc_2$ in $C_2$ do
11                 if $T[sc_1, sc_2] = \emptyset$ then
12                     $T[sc_1, sc_2] \leftarrow \sigma(sc_1, sc_2, T)$
13             end
14         end
15         Append($S$,$\tilde{\sigma}(C_1, C_2, T)$)
16     end
17 end
18 return Max($R$)
```

- `Length`(c) returns the number of words contained in the chunk c

- `Split`(c, i) returns a list of two chunks, one containing the words of c from the first to the ith, and the other containing the words of c form the $(i + 1)$th to the last.

  If the ith word is the last word of c, it returns a list containing the only element c.

- `Append`(L, x) is the standard *append* operation for lists, adding the element x in the last position of the list L.

- $\hat{\sigma}(C_1, C_2, T)$ returns the Chunk Score (see 3.3.2.4) of c_1, c_2 **given** a particular 2-split of the two chunks. Such a score is computed combining the ones of the smaller constituents of c_1, c_2, which are required to be already present in the table T.

  As it is described in Section 3.3.2.4, a number of features are employed to form a full chunk score, the main one being the **average** of the scores of the best alignments of the sub-chunks in C_1 with the ones in C_2.

  For example, given $C_1 =$ [“increase”,“the volume”] and $C_2 =$ [“raise”, “the volume”], “increase” is likely to achieve the best score with “raise” (eg.

T[“increase”,“raise”] = 0.8), as well as “the volume” will be aligned with “the volume” (T[“the volume”,“the volume”] = 1)). The score of “increase the volume” and “raise the volume”, in this particular splitting, will thus be the average of the two, 0.9.

After these premises it is relatively easy to walk through the pseudo-code of Algorithm 1. Lines 1 and 2 handle the base case, that is, when the algorithm is run on **two words**; in this case the result of the Word Similarity function, which is described above, is returned.

At line 3 the list of **candidate results** is initialized as an empty list. This is because more than one score will be computed for c_1 and c_2, the maximum of which will be returned as a result.

Lines 4 and 6 define a nested loop structure, which purpose is to update the two indexes i (from the first to the last word of c_1) and j (from the first to the last word of c_2) to cover **every possible combination** of word positions in the two input strings. These indexes are used to **split** the input chunks in smaller parts (lines 5 and 8). As an example, if the input is given by c_1 =“increase the volume”, c_2 =“raise the volume”, the outer loop will produce the three possible values of C_1: [“increase”,“the volume”], [“increase the”,“volume”] and [“increase the volume”]; each of these values will be combined, in the inner loop, with the three possible values of C_2: [“raise”,“the volume”], [“raise the”,“volume”] and [“raise the volume”].

Note that, while it is reasonable to consider the entire c_1 and c_2 in the comparisons (eg. if c_1 =“quit”,c_2 =“quit please”, we’d want the whole c_1 to be associated with the “quit” sub-part of c_2), it is necessary to prevent the two entire c_1 and c_2 to be associated at the same time, thus producing an **infinite loop**; this is done in line 7.

The next part of the inner loop, from line 9 to line 10 combines every sub-chunk of c_1 ($sc_1 \in C_1$) with every sub-chunk of c_2 ($sc_2 \in C_2$); note that C_1 and C_2 contain either one or two elements. Every combination that is not present in the table is scored with a **recursive** call to the algorithm itself, and the resulting score is saved in the table (line 12).

The last operation done in the loop, at line 15, is to combine the sub-scores that have just been scored in the table to compute the final score of this particular subdivision of c_1 and c_2. This is a **candidate score** for c_1 and c_2, and thus is appended to the list of candidate scores. As we already said, the definition of this combination is open; however it is most reasonable for the $\hat{\sigma}$ function to find the **best alignment** between the input sub-chunks, and provide a score based on it. It is clear that the definition of $\hat{\sigma}$ is crucial for getting sensible results from the algorithm. Furthermore, $\hat{\sigma}$ is the point where the algorithm can be extended with Machine Learning capabilities, that will be discussed in Section 4.2.

At line 18 the function returns the **maximum of the candidate** score, that is, the split that produced the best score (eg. for "increase the volume" vs "raise the volume", the score of "increase | the volume" vs "raise | the volume" is likely to produce a better score than "increase | the volume" vs. "raise the | volume"). Note that this decision determines the branching of one level of the recursion tree; once the algorithm is done computing the score between two sentences, a traceback of the selected splits can be interpreted as a parse tree for the two input sentences.

4.2 Machine Learning

The algorithm described in section 4.1 is mainly concerned with **static features** to measure similarities: the score of two words is derived from equality, edit distance, Wordnet path length and so on; the score of chunks is the average of the scores of the best alignments of their components.

In this section we address how does the algorithm **learn** from the sentences it is parsing. In other words, given a sentence of which I know the meaning label, how this information is used to improve the way I process new sentences.

Here I consider **three features** based on Statistical Language Processing methods to address this problem: chunk likelihood, class-conditional chunk weight and alignments likelihood. As it is summarized in Section 3.4, these features are used at every scoring level in the system.

4.2.1 Chunk Likelihood

Let's consider the following pair of sentences:

(8) Pump up the volume

(9) Turn up the volume

As an English-speaking human being I immediately associate the phrase “pump up” with “turn up” and “the volume” with its analogue. One of the reasons I make this association rather than, for instance, associating “pump” with “turn” and “up the volume” with its analogue, is that I have good experience of the phrase “the volume” being used in different contexts, while fewer times I encounter “up the volume”.

Thus a feature that can be useful to **enforce good chunking** of the input sentences is the likelihood of a chunk of text (where a chunk is defined as either a word or a combination of chunks). This feature can be modeled as the relative count of each chunk in the whole pool of the ones that have been recorded:

$$l(c) = \frac{\text{\# of } c}{\text{\# of total chunks}}$$

Note that this feature is completely unsupervised, in that it does not require a Meaning label to be computed.

It is also important to point out that, in order for this feature to be effective, a fair amount of training data is necessary. We encountered **data sparsity** problems when training this feature on our reduced corpus of meanings, and therefore in later experiments we decided to replace it with another language model based on corpus-trained parse trees. The purpose of this system is, for every possible chunk of the input sentence, to tell whether that chunk is a linguistic constituent or not of the sentence it is taken from. This is done according to the following procedure:

1. The entire input sentence is parsed using the CYK algorithm [CITATION] trained on the Penn Wall Street Journal corpus [CITATION]
2. Every possible constituent is extracted from the parse tree.
3. If the input chunk is one of the constituents, 1 is returned, 0 otherwise.

The result of this procedure is **multiplied** to the `AAVG` feature of Chunk Scores (and Word Scores as well), in order to set, in the final score, a preference for linguistically plausible alignments.

Note that the choice of performing this multiplication, rather than having the chunk likelihood as a stand-alone feature, is also beneficial in that keeps the chunk likelihood, which can be considered as a **language modeling** feature, from occupying part of the final mass, which should be considered as an alignment mass.

4.2.2 Class-conditional Chunk Likelihood

Let's consider the following three sentences:

(10) Increase the volume

(11) I would appreciate if you could increase a bit the the volume

(12) Decrease the volume

With a reasonable approximation, the first two sentences have the same meaning, while the last says the opposite, although the first and the last sentence are much more similar from a syntactic point of view. Nevertheless, no English speaker would bother wondering about hidden meanings behind phrases like "I would appreciate", or "if you could": everyone could tell that the second sentence is no more than the first one, with some more formal dressing.

Especially, when I listen to the second sentence, I can immediately locate "increase" and "the volume" as the most informative phrases, while filtering out the rest as less relevant. One of the reasons I am able to do this is that I have memory of "I would appreciate if you could" used in sentences conveying lots of different meanings, while "increase" is likely to appear only in situations where something is increased.

A feature that could be useful to spot the **informativeness** of chunks can be the conditional probability of the chunk, given a certain meaning:

$$l(c|m) = \frac{\#\text{ of } c \text{ in } m}{\#\text{ of } c}$$

This feature is used in **Meaning Scores** (see 3.3.2.2) to measure how much the terms in the input sentence are relevant to a certain meaning. Even though this is not done yet, it could also be used to weight the score of different chunks in the sentence, so for more informative chunks to influence more of the total score mass.

4.2.3 Alignment Likelihood

Let's consider the following pair of sentences:

(13) Skip this track

(14) Jump over this song

Again, English speakers can easily associate the meaning of "skip" with the one of "jump over", and the meaning of "track" with the one of "song".

One device that the algorithm uses to solve this problem is the **Wordnet** Path Lenght measure for word comparison. However, it has been noticed that this measure has its limits. In particular

- Not all the words are included in Wordnet
- Being a static measure, it cannot be adapted to the domain
- It has difficulties spotting antonyms

The alignment likelihood measure is used in Chunk Scores and Word Scores (see Sections 3.3.2.4 and 3.3.2.4) to **enforce correct alignments** of different chunks.

$$l(c_2|c_1) = \frac{\#\text{ of } c_1 - c_2 \text{ alignments}}{\#\text{ of } c_1 \text{ alignments}}$$

Note that, since no alignment labels are provided in the dataset (each sentence is labeled with a meaning, but its syntactic structure is hidden), this measure have to be trained in an unsupervised fashion. This problem shares a lot with the alignment likelihood in **Machine Translation**, even though here we can take advantage of the chunks sharing the same language (thus allowing some clever initialization of unsupervised methods with sensible guesses based, for instance, on Wordnet). On the other side our problem presents less precise training data, in that, while MT alignments are trained on sentence pairs, here we have a group of sentences sharing a same meaning; it can be argued that this cannot allow for strong syntactic assumptions.

4.2.4 Training

All the features presented in this section need to be trained on a corpus before being employed. It should be pointed out that our hand-made **corpus of meanings** is not big enough for good results to be expected from it, as it includes only six meanings, each one defined by a set of three or four sentences. As fare as we know, there is no existing corpus that could be used for this purpose. In Section 6.2.1 we propose a method to produce such a corpus, that would however require a separate future work.

The training of the **chunk likelihood** and **class-conditional chunk likelihood** parameters is trivial, as it only involves a relative counts of all the possible chunks derived from the existing example sentences.

The training of the **alignment likelihood** requires more effort, as the training data does not include information about chunk alignments. The natural way of training this parameter would thus be to run an Expectation-Maximization procedure [CITATION], that retrieves alignment hypotheses from the scoring algorithm, and uses these hypoteses to improve the parameters of the algorithm itself in an iterative way. Due to time constraint, this procedure was not implemented.

What happens in fact is that the algorithm is repeatedly run (a fixed number of times) on sentences labeled with the same meaning. As we have seen in 4.1, the algorithm produces a score for every possible alignment of the two input sentences. This score is used to increment the total recorded **alignment mass** of the alignment, that is then used to draw the statics described in Section 4.2.3. In this way we achieve an iterative update that is similar to the one performed with EM, but for the fact that the number of iterations is fixed, rather than dependant on EM's variation threshold.

The part of the software that is concerned with training is the `lu.learn` package; specifically, the procedure described above is contained in the `lu.learn.sentence` module.

4.3 Summary

In this chapter we described an algorithm to compare the similarity of two input sentences. Its main **advantages** are the concurrent consideration of many features like word meanings, position and overlapping, as well as the structure of the sentences, having the result in the form of aligned binary trees. The algorithm is recursive and exhaustive in that considers all the possible alignments of all the possible combinations of all the possible binary trees of the two input sentences, maximizing the efficiency with dynamic programming. Due to its modular structure, the algorithm can be easily extended with new features, allowing its integration with well-known Machine Learning techniques in Computational Linguistics (eg. Expectation-Maximization for computing word/phrase alignment probabilities).

There are some assumptions and **weak points** as well. First of all, the algorithm is inherently binary, in that every recursion step is made of a 2-split of the two input strings; this may have an effect in the way partial scores are combined. More generally, this “two times binary” recursive structure makes the features design task hard, since the effect of one operation is not easy to control over a potentially unlimited number of recursion steps. Also the number of parameters can become high, since each score is a linear combination of single features, requiring a separate Machine Learning training.

Lastly, it has to be pointed out that the algorithm we present, due to its extensible features system, is not meant to be definitive, but should rather be considered as a **backbone** for future extensions. The main steps ahead before expecting good results from it are described in Section 6.2.2.

Chapter 5

Interaction

In the previous chapter we have seen how the Language Unit is able to associate utterances from the user with their most likely meaning; due to the nature of the problem, it cannot be guaranteed that the output meaning is actually the one intended by the user. However, we have also seen that such an association is the result of a fuzzy matching process, where the utterance receives a comparison score with each of the candidate meanings. When these scores are accurate, they provide an important **estimate of the confidence** whether the understanding is correct or not.

This chapter describes how the Language Unit exploits the information derived from the matching scores, to decide on whether a grounding interaction with the user is needed, and in this case asking **pertinent clarification questions** about the meaning of single sentence components. Also, an explanation of the **learning** process is given, that is, how the LU makes use of the user answers to update and improve its knowledge about the language.

Section 5.1 describes how the best interaction policy is derived from the match scores; Section 5.2 explains how the software locates the minimal unclear constituents in the input sentence; Section 5.3 deals with how the Language Unit interact with TDM to produce the grounding subdialogues; Section 5.4 describes how the user answers are used to learn new information about the language.

5.1 Meaning Matching Workflow

The first problem the Language Unit has to face, is to determine whether the understanding of an input utterance is good enough to be taken as correct, or it rather **needs to be clarified** interacting with the user.

5.1.1 Overview

Let's consider the following situation:

```
S> How can I help you?
U> what is the title of this song
```

At this point, four different **scenarios** are possible:

1. There is one best matching meaning, and its confidence score is high.
2. There is one best matching meaning, but its confidence score is low.
3. There are two or more meanings with a high confidence score.
4. No meaning has a good confidence score.

These different situations can be easily associated with as many **interaction policies**:

1. Accept the best meaning without interaction.
2. Ask whether the understanding of the best meaning is correct.
3. Ask the user to disambiguate between the top candidates.
4. Ask the user to rephrase his utterance.

5.1.2 Implementation: the `get_plan()` Function

The work of **deciding on the policy** to adopt is done by the `lu.learn.interaction` module, and especially by the **`get_plan()`** function. This function takes a list of candidate meanings (and their confidence scores) as input, and outputs the plan that the dialogue manager should follow. In the case of TDM, the `get_plan_tdm()` function can be called, which returns the plan in the form of TDM-compatible dialogue moves[1].

At first, `get_plan()` **normalizes** the input scores, so that they sum up to 1. This allows us to compare them scores more easily[2].

When the scores are normalized, **confidence thresholds** can be set in order to detect the four cases showed in 5.1:

1. The best score is greater than a constant α. The value of α should be chosen so that $\alpha > 0.5$; in this way it is guaranteed that at most one meaning will be selected with maximum confidence.
2. There is one and only one meaning whose confidence score is greater than a constant β, and yet smaller than α. The value of β should be chosen so that $\beta < \alpha$.
3. There are two or more meanings whose confidence score is greater than β and smaller than α. In order to allow for this possibility, β should be chosen so that $\beta < 0.5$
4. None of the confidence scores is greater than β.

In the experiments, the α and β **constants** have been manually set to, respectively, 0.7 and 0.3. As it will be better discussed in Chapter 6, a better way to select these parameters would be to employ a meta-learning procedure to train the hypermodel for optimal performance.

At this point, **cases 1 and 4** are trivial to handle: `get_plan()` will return the best matched meaning in the first case, and the meta-goal `_rephrase`[3] in the latter. `_rephrase` has the effect of terminating the interaction with the system utterance

(15) I do not quite understand what you mean, can you rephrase please?

[1]Due to the lack of documentation of the TDM code, it was not possible to integrate the grounding moves into the dialogue management library. Instead, a generic entry-point-move is used to start grounding subdialogues that are in fact controlled by the Language Unit. More details will be given in 5.3.

[2]Note that, even if this is not done in the current implementation, one could think of using the un-normalized values to have an idea of the absolute confidence of the matches.

[3]Note that, since `get_plan()` is called by TDM to interpret user utterances, formally this means "User asked System to ask User to rephrase". A more elegant way of solving this issue would be to integrate rephrase requests in TDM, but this, again, was not possible due to lack of documentation.

The following dialogue shows SVPlay dealing with the two situations that have just been described:

```
S> How can I help you?
U> what is the current song
S> Running too fast, by Rehearsal Summer
S> How can I help you?
U> I like biscuits!
S> I do not quite understand what you mean, can you rephrase please?
```

Cases **2 and 3** require a more sophisticated handling. In this cases, the Language Unit will first try to locate specific parts of the utterance that are particularly unclear (see 5.2), and then use them to produce a targeted clarification request to ground the correct meaning of the user utterance (see 5.3).

5.2 Ambiguous Fragment Location

When the Language Unit faces uncertainty in the interpretation of the user utterance, a procedure is used to reduce the uncertainty to the **smallest possible constituents** of the sentence. This is done by analyzing the similarity between the user utterance and the best matching sentence within each of the candidate meanings.

5.2.1 Overview

Let's consider, as an example, the meaning `ask(?X.current_song(X))`, defined by the sentences in 16, and the question in 17. If this question matches this meaning, achieving its best score with the sentence in 16a, the software should then return "the current" from 17 and "this" from 16a as the minimal uncertain alignment, because all the other parts are **perfectly matched**. The way this especially unclear constituents are detected will de described in 5.2.2.

(16) a. What is the current song

b. Which song is playing

c. Which track is playing

(17) What is this song?

Of course, the algorithm should have a **margin of tolerance** as to the constituents whose match should be considered accepted. As an example, the sentence in 18 shouldn't raise concerns about the "What's"/"What is" alignment, when compared against 16a; this is because, even though the match between "What's" and "What is" is not perfect, it is expected to achieve a sufficiently high score to be considered matched without questions.

(18) What's this song?

5.2.2 Implementation: the `locate_fragments()` Function

Algorithm 2 defines the `locate_fragments()` function, which is used to locate, in the utterance from the user, the **constituents** that are especially unclear.

Algorithm 2: The minimal ambiguous fragment location algorithm

```
Procedure locate_fragments()
    if $s$ is WordScore then return $s$.from, $s$.to;
    $s_1 \leftarrow$ s.get_alignment(1).get_score();
    $s_2 \leftarrow$ s.get_alignment(2).get_score();
    $s_{1_N} \leftarrow \frac{s_1}{s_1+s_2}$;
    $s_{2_N} \leftarrow \frac{s_2}{s_1+s_2}$;
    if $|s_{1_N} - s_{2_N}| > \gamma$ then
        return locate_fragments(min($s_1, s_2$))
    else
        return $s$.from, $s$.to
    end
```

`locate_fragments()` is a **recursive** function, that receives a Sentence Score s as input[4] (it will be fed with the score between the user utterance and the best matching sentence in every candidate meaning of it), and outputs the two minimal constituents for which the alignment is uncertain (e.g. respect to the example mentioned in 5.2, it is expected to output the two chunks "the current" and "this"). This information will be used to produce **clarification questions** for the user (e.g. "Do you mean 'the current' when you say 'this'?").

[4]Note that, both here and in the actual implementation, a Score is not just a numeric value, but also contains information about the matched elements. For instance, scores define the fields `from` and `to`, containing the two elements (sentences, chunks, or words) the score refers to.

We can see at **line 2** that one reason that ends the recursion is when a `WordScore` object is given as input. This is clearly because no smaller constituents can be found in a score between two words.

If the input score is not a `WordScore`, it must then be a `ChunkScore`[5]. We know from Chapter 4 that every score between two chunks builds on two scores of smaller chunks; for instance, the score between sentences 17 and 16a, σ_S(“What’s this song”,“What is the current song”), will ideally contain the scores σ_C(“what’s”,“what is”) and σ_C(“the current song”,“this song”). **Lines 3 and 4** retrieve the scores of these two smaller alignments from the main score object.

Lines 5 and 6 normalize the inner alignment scores so that they sum up to 1. This is to uniform the values the algorithm will deal with in the next steps.

The final `if` construct (**lines 7 to 11**) looks at the difference between the scores of the inner alignments: if there is a significant difference between these two scores (γ is a free parameter) then repeat the algorithm on the weakest alignment (because it means that it contains the misunderstood part, while the other was understood with good confidence), otherwise return the whole chunks contained in the input score (because it means that both the alignments are somehow weak).

5.3 Production of the Interaction Subdialogue

Once the need for clarification is established (5.1), and the software has located the uncertainty (5.2), it is necessary to produce a subdialogue with the user, to put the **clarification plan** into practice.

5.3.1 Overview

In 5.1 we have encountered **two cases** in which an interaction with the user is necessary to ensure the correctness of the interpretation:

- The best matched meaning did not achieve enough confidence to be taken as correct

[5]It could also be a `SentenceScore`: this happens in the first call to the algorithm. However, in the current implementation, the `SentenceScore` class is just an alias for `ChunkScore` (see 3.3.2.3)

- More than one meaning achieved an equally high level of confidence

Even though these two situations could be approached in two different ways, only **one grounding subdialogue** has been designed. and that is identified by the label `_ground`.

This episode, due in part to technical restrictions (as it will be explained in 5.3.2), is very simple, and consists of a single question, that can be answered affirmatively or negatively by the user. The following is an example of a `_ground` episode:

```
S> How can I help you?
U> what is the title of this song
S> Does ‘title of this song’ mean ‘current song’?
U> yes
S> Thank you for the feedback!
S> Young Lust, by Pink Floyd
```

Note that the **clarification question** can be any string, and can make use of the information returned by `locate_fragments()`, that is, the two chunk of text whose alignment is uncertain. In the current implementation, this question is always given in the form of “Does X mean Y”, where X and Y are the chunks returned by `locate_fragments()` (see 5.2); different formulations could be thought, to obtain a more natural interaction with the user (e.g. simply asking “You mean ‘current song’?”).

If there is only **one candidate** meaning (case 2 in 5.1), the question will be referred to that meaning, **otherwise** (case 3 in 5.1) the question will be about the meaning that achieved the best score. Note that in fact this mean that the two cases are **not distinguished** in any way from each other.

However it is also worth to note that the full plan produced by the Language Unit, even though not implemented because of technical limitations, **does differenciate** the two situations, and includes alternative questions for more precise disambiguation.

5.3.2 Implementation: the `_ground` plan

As it has already been mentioned, the TDM library comes with no technical documentation. This unfortunately limited us to the design of extremely simple grounding interactions. For the same reason, the design and implementation of such interaction, which is described in this section, is to be considered as a substantial **workaround**, wich does not fit the design principles of neither TDM or LU: an integral rewrite should be considered upon availability of TDM's documentation.

Every time the `get_plan()` function of the **Language Unit** (see 5.1.2) encounters one of the situations mentioned in 5.3, instead of returning an ordinary dialogue move, it performs the following operations:

1. Pushes the uncertain meaning in a stack of open grounding issues. This stack is defined in `lu.learn.interaction` as `tdm_ground_stack`.
2. Returns the `request(_ground)` meta-move to TDM, that called the LU in the first place. This call is made from `tdm.lu_grammar`, by the method `utterance_to_moves` of the class `LuGrammar`, which is itself called by method `interpret` of class `InterpretModule`, defined in `tdm.interpret`. Along with the grounding request, the LU also sends the English question that will be asked to the user.

The following actions are performed by **TDM** upon receival of the `request(_ground)`:

1. Changes the realization string for the meaning `ask(?X._ground_X_to_Y(X))` (which will be later used to ask the question to the user) into the question provided by the LU.
2. Executes the normal plan associated with `_ground`.

The **`_ground` plan** is made of two parts:

1. `findout(?X._ground_X_to_Y(X))`, where the system question generated by the Language Unit is answered with yes or no.
2. `dev_perform(Ground, MplayDevice)`; the `Ground` device action is triggered to handle the answer to the previous question.

The **`Ground` device** action terminates the interaction with the user. If the answer was negative (X does not mean Y), grounding was not successful and the system simply concludes the interaction asking the user to rephrase its utterance. Otherwise, if the answer was positive, the following grounding operations are performed:

1. Contacts the Language Unit to solve the top open grounding issue. This will pop the top element of the stack of open grounding issues, and possibly start the learning process.
2. Execute the action the user wanted in the first place.

The second point is somehow tricky, in that there is no known way for the device to interfere with the dialogue management operations. The solution to this problem is particularly inelegant: the device imports the Turn Manager, wich has been extended with a `ground_hack()` method. This method has the effect of posting the ad-hoc `GROUND_HACK` event in the system, with attached the meaning label of the action to run. A handler for this event has been written in the `tdm.interpret` module, which receives the meaning string, parses it, and finally throws the `INTERPRETATION` event that will run the action.

5.4 Knowledge Update

Every time a new labeled example (sentence + meaning label) is encountered, the Language Unit updates its knowledge about the language. In the current implementation, learning a **new sentence** consists of

- Adding the **new example** to the set of the sentences that realize its meaning
- Updating the global **chunk count** (see 4.2.1) for every possible substring of the input sentence
- Updating the **class-conditional chunk count** (see 4.2.2) for every possible substring of the input sentence
- Updating the **alignment mass** (see 4.2.3) for the alignments that are found parsing the new utterance with the ones already existing realizing its meaning.

These operations are done by the `lu.learn.sentence` module, and especially by its `learn()` method.

Adding the new example to the meaning definition is trivial, and incrementing the counters is straightforward as well: a procedure loops through every possible substring of the new sentence, and increments the corresponding global and class-conditional counter. Nevertheless, it is worth to spend some words on the last one, the **alignment mass update**. To update this value, the new sentence is scored against all the other sentences in the meaning; as we have already seen, every Score object carries information about the internal alignments of the two sentences (i.e. a Score is either a `WordScore`, or is built upon two smaller scores). What the learning procedure does is looping through this boxed scores, and for each of them adds its numeric value to the alignment mass already occupied by that score.

It is also useful to remember that, at the moment of learning, the user gave us information about the most **uncertain alignment** in the sentence (see 5.2), confirming that it is a valid alignment. This information is used to enhance the learning process, because, without the user answer, that alignment would receive a small mass increment, as its score is not expected to be high. But, considering the user input, the alignment can be reinforced by a significant value (e.g. 1), increasing the probability of it being recognised correctly in other contexts.

5.5 Summary

In this chapter we have described the way the Language Unit deals with potentially misinterpreted utterances.

The first step of the process is to **detect** whether the interpretation contains uncertainty or not. This is done by looking at the confidence score of the interpretation; four cases can be identified: no uncertainty, uncertainty with one candidate meaning, uncertainty with two or more candidate meanings, uncertainty with no candidate meanings.

When there is uncertainty and at least one candidate meaning, the software will proceed to **locate** the uncertainty, that is, finding the most specific possible constituent in the user utterance that caused the misinterpretation. This is done analyzing recursively the components of the score the input sentence achieved with the most similar example in each of the candidate meanings.

The next step will be to start a **clarification** subdialog with the user, to determine whether the uncertain alignment was legit or not. This dialog is always given in the form of a single modal question.

When the answer to the clarification question is positive, meaning that the software guess on the interpretation was correct, the software **learns** from the episode, adding the new sentence to the realizations of its meaning, and especially reinforcing the uncertain alignment, in order for it to be recognised in further situations.

Chapter 6

Conclusions

6.1 Contribution of This Project

We have presented **SVPlay**, a music player application that is able to process **natural language** commands and deal with uncertainty in unknown input sentences, by performing fuzzy **matching** at meaning level, interactive **clarification** with the user (when necessary), while **learning** from both matching and interaction.

In particular, we have built a **client application** for OpenTDM, a dialogue management library that implements the Information State Update (ISU) framework; the client application (see 3.1) defines terms (ontology), dialogue plans (domain) and actions (device) that are used by the player, and a language file, that defines how the other elements are referred to in English. We designed and implemented the **Language Unit** (LU, see 3.3), a drop-in replacement for OpenTDM's language understanding and generation modules (see Section 1.2), that is able to match the meaning of unknown input examples. We modified OpenTDM at **dialogue management** level, to include grounding and clarification capabilities to the system.

We designed the **matching algorithm** of the LU (see Chapter 4) taking into account the following principles:

- Matching is based on **nested scores** comparisons, to take into account similarity at different levels. In fact, the score between a sentence and a meaning involves scores between the sentence and other sentences labeled with that meaning; sentence scores in turn are based on scores between the constituents (chunks) of the two sentences, and the chunk scores rely on scores between single words.

- Scores are always expressed as linear combinations of independent **extensible features**, in a way that is inspired by IBM's work on Watson (see Section 2.1). This allows for future extensions of the comparisons with more features, and favoures developments at meta-model level (see Section 6.2.2).
- Comparisons are **data-driven**, in that feature exist, at every level, that take into account data statistics in an all-fragment fashion, letting similarities and structures emerge from the existing examples, and evolve when new examples are presented.

The **clarification** interaction features a policy selection procedure, that evaluates the need for interaction on the basis of the match confidence scores (see Section 5.1), and an algorithm to locates, as specifically as possible, the input sentence constituents which interpretation was uncertain (see Section 5.2). When a case of misinterpretation is solved through the interaction with the user, the information provided by the latter is used to **update** the system's language knowledge, and used to improve the interpretation accuracy on new examples (see Section 5.4).

6.2 Future Work

Even though all the functionalities described in this thesis are implemented and working in SVPlay , the software we produced is still to be considered as a **prototype**. This section illustrates the steps that separate SVPlay from the release stage, along with some extra improvements that could be thought for the future.

6.2.1 Evaluation

A first step to take before planning software improvements would be to evaluate its performance at the current stage. This turned out to be a particularly difficult task, because of the nature of the task we are evaluating. To be more precise, there are **two main tasks** in SVPlay that should be evaluated; effectiveness at the level of dialogue system and accuracy at the level of meaning matching.

Jurafsky and Martin (2008) suggest three classes of metrics to evaluate a **algorithm**: user satisfaction, task completion cost and task completion success. All of these three classes of metrics requires a study based on the user interaction with the system; while user satisfaction metrics are based on the subjective opinion of users, the other two take into account statistics on the tasks that are being completed: task completion cost metrics measure negative features of the interaction, such as completion time, number of queries, and so on; task completion success measures draw statistics on the completed tasks (percentage of completed task, correctness of the single answers etc.).

The accuracy of the **meaning matching** algorithm of SVPlay could be evaluated with standard, corpus based, precision and recall statistics. That is, given a corpus of sentences, where each sentence is associated with a meaning label, train the Language Unit's meaning matcher on a training subset of the sentences, and get its classification hypotheses on the remaining, testing, part of the corpus. The labels of the testing sentences can be used as a gold standard to produce accuracy statistics.

The main obstacle against corpus based evaluation is that such a corpus, as far as we know, does not exist. Nevertheless, gamification and crowdsourcing could be used to gather a user-generated corpus of sentences to fit this purposes. This is because of the analogies between the task of building the corpus and the famous game ***Taboo***. In this game, the players have to find ways to describe a certain concept to each other, without making use of certain words or expressions related to the concept. An online version of this game can be thought, where the conceps are meaning labels, and the expressions to be avoided are sentences already present in the dataset. Players could compete with each other, gaining more points the more popular their answers are among other players. When the popularity of an expression overcomes a certain threshold, it is added to the dataset, thus filtering out the spam coming from the less cooperative users.

6.2.2 Accuracy improvements

Another area where SVPlay could be improved is the **matching accuracy**. Here we suggest some ways to improve the quality of the matches returned by the Language Unit.

First of all, as we have explained in 3.3.2, every score in the system is a linear combination of single features. The **weights of this linear combination** are a part of the *hypermodel* of the scoring algorithm that is particularly suited to machine learning training (as it already happens in Watson). An automatic way of optimizing these weight would be beneficial in that would improve the performance of the system as it is, and would allow the definition of a greater number of comparison features, without the risk of losing control of the model parameters. However, machine learning optimization of feature weights requires a labeled corpus of sentences, chunks and words. A way of gathering such a corpus was presented in Section 6.2.1.

It could also be thought of employing **dynamic weights** for matching, that is, to adapt the feature weights depending on other comparison features (one example would be to disregard all the other Word features when the Equals feature is 1).

Furthermore, we believe that a stricter application of the TF-IDF principles (see section 2.1) would provide great accuracy improvements. At the moment the software uses a *class-conditional likelihood* feature for meaning scores (see 3.3.2.2); this feature is an analog of Term Frequency, as it represents the frequency of chunks from the input sentence in the examples that define the meaning. An analog of **Inverse Document Frequency** would prevent generally common terms to influence the scores as much as meaning-specific terms do. As an example, let's consider the case of sentence 19 being compared with sentences 20 and 21. The algorithm will devote a lot of score mass to the similarity of the "What's" chunk, even though its common use makes it irrelevant de facto in the comparison. A IDF measure would prevent this from happening.

(19) What's this track?

(20) What's your name

(21) What's the current song

When a IDF measure is on place, it could be thought of performing a **multi-pass** matching to gradually restrict the list of candidate meanings. As an example, let's consider the case of sentence 22, matched against sentences 23, 24 and 25. Given that the software has no alignment between "Turn up" and "Increase", the chunk "the volume" would be useful at first, to restrict the candidate matches to 23 and 23. At this point, running the algorithm a second time would have the effect of reducing the IDF of “the volume” to zero, forcing the system to focus on the comparison between “Turn up”, “Increase” and “Decrease”.

(22) Turn up the volume

(23) Increase the volume

(24) Decrease the volume

(25) What’s the current song

We have seen in Section 5.4 that the user’s positive feedback on clarification requests is used to learn new alignments between chunks in the language. However, no learning is performed when the feedback is negative (that is, when the user states that an alignment between chunks of text is not lecit). **Negative feedback** could be used to improve the model, for instance reducing the score mass associated with the alignment the user said to be wrong.

Also, when a new alignment is learnt between chunks c_1 and c_2, it can be thought of **propagating** the new similarity information to the other chunks c_i for which $\sigma_c(c_1, c_i)$ is sufficiently high. This would allow for the definition of fuzzy synonym sets where the transitive property of the synonymy relation.

Finally, one of the problems of the current implementation of the alignment features is that the alignment mass of very frequent chunks/words keeps being reinforced, thus becoming disproportionate respect to the others. The result of this is that some alignments are scored less than others, just because the concept they express is less frequent than the concept expressed by others. One way to reduce this problem would be to implement some sort of **economy** of score masses, where, when needed, an increment of an alignment mass causes the decrement of other value, with the purpose of keeping the values equilibrate, and discouraging improbable alignments.

6.2.3 Efficiency improvements

One of the issues with the current implementation of SVPlay is the time required for the system to provide the answers. This is because the matching algorithm is not yet optimized for optimal performance. In this section we propose some ways to **speed up** the meaning matching process.

We have already mentioned that meaning matching depends on sentence matching, and the latter is an *all-fragments* process: every possible combination of strings from the input sentences is taken into account during the comparison. This is not always necessary, as sometimes the **degree of confidence** of the alignment between two chunks is high enough to make other alignments options unuseful. A limit case of this is represented by the alignment between identical chunks: at the moment this case is not distinguished from alignments between different chunks, where it would make sense to directly align identical chunks before matching the remaining ones.

Also often, during the comparison of two sentences, fragments of the sentences are encountered that have been scored before. A **cache** for these partial score is likely to save a significant amount of computation time.

Another way of improving the speed of meaning matching would be to exploit the **context** of dialogue: at the moment no information is given from the dialogue manager to the LU circa the range of possible meanings that the user may express at a specific point in the conversation. If this information was given, it could be used to reduce the comparisons to the meanings that are indeed possible. As an example, if, at a certain point in the dialogue, the user is supposed to answer the question "Do you want to quit?", the LU could match the next user utterance against the only two meanings `answer(yes)` anb `answer(no)`.

Finally, a **coarse-to-fine** matching procedure could be devised, that first restricts the set of possible meanings using computationally inexpensive features (e.g. TF-IDF measures alone), and then runs the full matching algorithm on this restricted set to find the correct answer.

6.2.4 Extra features

In addition to performance improvements, we propose here some features that could naturally **extend the software** as it is now.

First and foremost, the **OpenTDM integration** should be improved, upon availability of its technical documentation. This would allow for the creation of more sophisticated and natural dialogues with the user, especially in the grounding case that we have described in Section 5.3.2.

A next step in the development of the Language Unit would be the inclusion of parametrized sentences. As an example, let's consider the sentence in 26. At the moment the software could not support this kind of sentences, as it is not possible to associate part of the input with external parameters. This functionality could be integrated by making it possible to define models for **parameter types** in the client application. Sentence 27 shows how a parametrized sentence could look like; parameters models are needed to allow the LU to tell how likely a chunk of text is to belong to type `TITLE` or `AUTHOR`.

(26) Play the Ninth Symphony by Beethoven

(27) Play `<X:TITLE>` by `<Y:AUTHOR>`

Another interesting step to take would be to include models for **meaning compositionality**. For instance, a user may want to say something like “Skip this song and decrease the volume”; the software should then be able to recognise that the meaning expressed by this utterance is a combination of the meanings of its two parts.

In the normal dialogue interaction, participants often keep track of the objects that are being referenced across different moments of the conversation. A **rich context** representation would be useful in automatic systems as well; as an example, let's consider the following dialogue:

```
U: Play California Dreaming
S: Ok
U: Also, put it in my Favourites playlist
D: Done adding ‘California Dreaming’ to the Favourites playlist
```

A dialogue like this would not be possible at the moment, as the software, by default, erases any common ground every time an interaction ends. Note that this kind of references would also be useful on the meta-level,for instance allowing the user to correct the system interpretation of previous sentences.

A final extension proposal for SVPlay is concerned with the fact that, at the moment, all the interaction subdialogues are predefined and static (i.e. every dialogue plan produces a specific sequence of invariable system utterances). It would be interesting (given the availability of a proper corpus) to apply a data-driven, **machine learning** approach to the **meta-level** of the interaction, being the structure of the dialogues between the system and the user, achieving human-machine conversations that are even closer to the natural human ones.

Appendix A

An Information State Update Example

This appendix shows how SVPlay's Information State (IS) is updated during a **simple user interaction**, where the latter asks for the current song title. The interaction, at input/output level, is the following:

```
S> How can I help you?
U> what is the current song
S> It's All Over Now, by The Rolling Stones
```

As a premise, it is necessary to point out that, as OpenTDM comes with **no documentation**, the information contained in this appendix was derived from our analysis of the software's source code and log files. For this reason, it is possible that our explanation will not cover some parts of the process, either irrelevant or still obscure.

At t=0 the information state of the application is **empty**. While for the next steps we will only consider a subset of the OpenTDM's Total Information State (TIS), Figure A.1 shows the complete TIS, as it was reconstructed from the application's log. We can immediately spot, in the figure, the two main parts of the IS: private and shared. The most relevant parts of the **private** are

- The list of the system's private **beliefs** (`bel`)
- The stack of **plan** items that the system will execute.

```
application             Application('svplay', <Ontology>, <Domain>)
device_outputs          stack([])
devices                 {'MplayDevice':  <Device>}
input_event             Event(START)
latest_moves            open_queue(['#'])
latest_speaker          None
                        [ alts        []                 ]
next_utterance          [ context     {}                 ]
                        [ moves       open_queue(['#'])  ]
                        [ plan_item   None               ]
output
passive_mode            False
                        [ agenda   open_queue(['#'])  ]
PRIVATE                 [ bel      {}                 ]
                        [ nim      open_queue(['#'])  ]
                        [ plan     stack([do(top)])   ]
program_state           RUN
recognised_utterance    None
                        [ actions           stackset([top])              ]
                        [ com               {}                           ]
                        [ issues            stackset([])                 ]
                        [                   [ moves        {}    ]       ]
SHARED                  [ lu                [ speaker      None  ]       ]
                        [                   [ turn_count   {}    ]       ]
                        [ pm                {}                           ]
                        [ previous_action   None                         ]
                        [ qud               stackset([])                 ]
sys_truns               0
timeout                 [ duration   None  ]
                        [ enabled    False ]
```

FIGURE A.1: An empty Total Information State in OpenTDM

From the **shared** part of the IS is important to highlight the stack of **questions under discussion** (qud), representing the issues that user and system will cooperatively try to solve. For the sake of record, we report that the lu field contains the *last utterance* to be pronounced, either by the system or by the user; similarly, pm stands for *previous move*, and com for *common ground.*

We notice that Event(START) is pushed by default in the input_event fileld (application and device_outputs are filled as well, but they just hold static information, that will not change during the execution). This is because OpenTDM is **event-driven**: everything happening in the system is a consequence of an event, and START is the one that starts the causal chain. The complete list of events is defined in the maharani.event module.

As a consequence of the START event. the software will plan his first dialogue move. This is done by pushing the top plan in the **private plan** section. The top plan is defined in the application ontology;

```
["forget_all",
 "findout(?set([issue(?X.current_song(X)), issue(?X.my_name(X)),
                action(increase_volume)]))"]
```

This clears all the current believes and raises the issue of determine **what the user wants to do**. Literally, the findout() statement aims to decide whether the user wants to know the title of the current song, know the system's name or increase the volume. As a matter of fact, SVPlay supports more than these three actions, but the logic form of the top plan could not be changed accordingly because of the lack of documentation.

This plan has the effect of producing a SYSTEM_MOVES event, that will prompt the user with the **question** "How can I help you?". Note that the surface form of this question will be provided by the Language Unit, and is defined in language/<APP><LANG>Prod.py, in the application directory. Once the question is asked, it can be pushed in the "questions under discussion" part of the **shared** IS.

The following is the resulting Information State:

```
input_event            Event(SYSTEM_MOVES,
                       [(Move(ask(?set([issue(?X.current_song(X)),
                                        issue(?X.my_name(X)),
                                        action(increase_volume)])),
                            speaker=SYS, modality=speech),
                            'How can I help you?')])
latest_speaker         SYS
                       bel   {}
PRIVATE                plan  stack([findout(?set([issue(?X.current_song(X)),
                                              issue(?X.my_name(X)),
                                               action(increase_volume)]))])
recognised_utterance   None
                       com   {}
SHARED                 qud   stackset([?set([issue(?X.current_song(X)),
                                            issue(?X.my_name(X)),
                                             action(increase_volume)]))])
```

At this point, the **user** inputs the sentence "What is the current song". This input is given to the language unit, that will return its formal interpretation, ask(?X.current_song(X)).

OpenTDM receives this interpretation (in the `tdm.interpret` module), and triggers a `USER_MOVES` event.

This event causes the system to look for a **plan** in the domain that matches this user move. This plan exists, and its content is

```
dev_query(?X.current_song(X), MplayDevice)
```

Which simply means to **query** the device for the title of the current song. Note that this new issue is also pushed in the shared questions under discussion.

The Information state is now the following:

```
[ input_event            Event(USER_MOVES,                                           ]
[                        open_queue([Move(ask(?X.current_song(X)),                   ]
[                                       speaker=USR, modality=speech), '#']))        ]
[ latest_speaker         USR                                                         ]
[ PRIVATE                [ bel    {}                                          ]      ]
[                        [ plan   stack([device_query(?X.current_song(X))])   ]      ]
[ recognised_utterance   What is the current song                                    ]
[                        [ com    {}                                           ]     ]
[                        [ qud    stackset([?X.current_song(X)]),              ]     ]
[ SHARED                 [                  ?set([issue(?X.current_song(X)),   ]     ]
[                        [                        issue(?X.my_name(X)),        ]     ]
[                        [                        action(increase_volume)]))   ]     ]
```

The device answer to the query has the effect of creating a new **belief** in the private part of the IS:

```
current_song('It's All Over Now, by The Rolling Stones')
```

Which trivially states author and title of the song that the device is currently playing. This information is **output** with a `SYSTEM_MOVES` event[1].

After the system answer, the "current song" issue is removed from the shared questions under discussion[2], and the answer is added to the **common ground** (being the set of propositions that are considered to be in the beliefs of both the dialogue partners), along with the fact itself that the question has been answered.

The following is the information state after the system answer.

[1]Note that here the language unit receives the logical form `answer(current_song('It's All Over Now, by The Rolling Stones'))`, to be turned into a English sentence. In this case the logical form, that comes from the device, contains a complete English realization. A more elegant way of solving the problem would be to have the device output just the song name and author, and let the LU form the sentence.

[2]Here the system also removes the top plan from `qud`; the reasons behind this choice are not clear.

```
input_event             Event(SYSTEM_MOVES,
                        [(Move(answer(current_song('It's All Over Now, [...]')),
                                         speaker=SYS, modality=speech),
                                         'It's All Over Now, by The Rolling Stones')])
latest_speaker          SYS
PRIVATE                 [ bel    current_song('It's All Over Now, by [...]') ]
                        [ plan   stack([])                                   ]
recognised_utterance    What is the current song
                        [ com    {current_song('It's All Over Now, by [...]'),  ]
SHARED                  [                     resolved(?X.current_song(X))}    ]
                        [ qud    stackset([])                                   ]
```

After the felicitous utterance, the system will run out of plan elements to execute, and will thus load the `top plan` again.

Bibliography

Joseph Weizenbaum. Eliza, a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966. ISSN 0001-0782. doi: 10.1145/365153.365168. URL `http://doi.acm.org/10.1145/365153.365168`.

Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language.* PhD thesis, February 1971.

Kristiina Jokinen and Michael F. McTear. *Spoken Dialogue Systems.* Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2009.

Stephanie Seneff and Joseph Polifroni. Dialogue management in the mercury flight reservation system. In *Proceedings of the ANLP-NAACL 2000 Workshop on Conversational Systems*, ConversationalSys '00, pages 11–16, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1605285.1605288`.

M Gabsdil. Clarification in spoken dialogue systems. In *In AAAI*, 2003.

Raquel Fernández, Staffan Larsson, Robin Cooper, Jonathan Ginzburg, and David Schlangen. Reciprocal learning via dialogue interaction: Challenges and prospects. Proceedings of the IJCAI 2011 Workshop on Agents Learning Interactively from Human Teachers (ALIHT 2011), 2011.

Palakorn Achananuparp, Xiaohua Hu, and Xiajiong Shen. The evaluation of sentence similarity measures. In *Proceedings of the 10th international conference on Data Warehousing and Knowledge Discovery*, DaWaK '08, pages 305–316, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85835-5. doi: 10.1007/978-3-540-85836-2_29. URL `http://dx.doi.org/10.1007/978-3-540-85836-2_29`.

Satanjeev Banerjee and Ted Pedersen. Extended gloss overlaps as a measure of semantic relatedness. In *In Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 805–810, 2003.

Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: parameter estimation. *Comput. Linguist.*, 19(2):263–311, June 1993. ISSN 0891-2017. URL `http://dl.acm.org/citation.cfm?id=972470.972474`.

Rens Bod. Unsupervised parsing with u-dop. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 85–92, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1596276.1596293`.

David A. Ferrucci. Ibm's watson/deepqa. *SIGARCH Comput. Archit. News*, 39(3):–, June 2011. ISSN 0163-5964. doi: 10.1145/2024723.2019525. URL `http://doi.acm.org/10.1145/2024723.2019525`.

David A. Ferrucci, Eric W. Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John M. Prager, Nico Schlaefer, and Christopher A. Welty. Building watson: An overview of the deepqa project. *AI Magazine*, 31(3):59–79, 2010. URL `http://dblp.uni-trier.de/db/journals/aim/aim31.html#FerrucciBCFGKLMNPSW10`.

David Traum and Staffan Larsson. The information state approach to dialogue management. In *Current and New Directions in Discourse and Dialogue*, pages 325–353. 2003.

Jonathan Ginzburg. *The Interactive Stance*. Oxford University Press, New York, 2012. URL `get-book.cfm?BookID=59315`.

Staffan Larsson, Staffan Larsson, and Publically Defended In Lilla Hörsalen. Issue-based dialogue management. Technical report, 2002.

H.H. Clark. *Using Language*. Cambridge University Press, May 1996. ISBN 0521567459.

Aarne Ranta. Grammatical framework: A Type-Theoretical grammar formalism. *Journal of Functional Programming*, 14(02):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/s0956796803004738. URL `http://dx.doi.org/10.1017/s0956796803004738`.

Staffan Larsson and David R. Traum. Information state and dialogue management in the trindi dialogue move engine toolkit. *Nat. Lang. Eng.*, 6(3-4):323–340, September 2000. ISSN 1351-3249. doi: 10.1017/S1351324900002539. URL `http://dx.doi.org/10.1017/S1351324900002539`.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

Edward Loper and Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1118108.1118117. URL `http://dx.doi.org/10.3115/1118108.1118117`.

George A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11): 39–41, November 1995. ISSN 0001-0782. doi: 10.1145/219717.219748. URL `http://doi.acm.org/10.1145/219717.219748`.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 2 edition, 2008. ISBN 0131873210. URL `http://www.amazon.com/Language-Processing-Prentice-Artificial-Intelligence/dp/0131873210%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0131873210`.