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

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence
ASR	**A**utomatic **S**peech **R**ecognition
CA	**C**onversational **A**gent
DM	**D**ialogue **M**anager
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 **gestures** and physical actions.

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. To this respect, the studies on Natural Language Processing (NLP) assume a dramatically central role, as dramatically central is natural language in the interaction of humans with each other.

For this reason, the focus of this thesis work is the area of **dialogue systems**. A spoken dialogue system, or conversational agent (CA), 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.

1.1 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.

1.2 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.

This document 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 ......

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.

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 to 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 of 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 the 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.

Accordingly, another 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), which 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.

Particularly inspiring for the development of this thesis was the work done by IBM on Watson. Watson is . . .

CL&DOP . . .

2.3 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; 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.

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 (Traum and Larsson, 2003).

2.3.1 Information State Update Dialogue Management

. . .

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 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.
- 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.

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

. . .

[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.

[2]http://www.grammaticalframework.org/

[3]http://talkamatic.se/

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. 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.

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

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

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.

- `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.

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

- `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 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.

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

[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.

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.

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

  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.

- `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 **four 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.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

[ht]

. . .

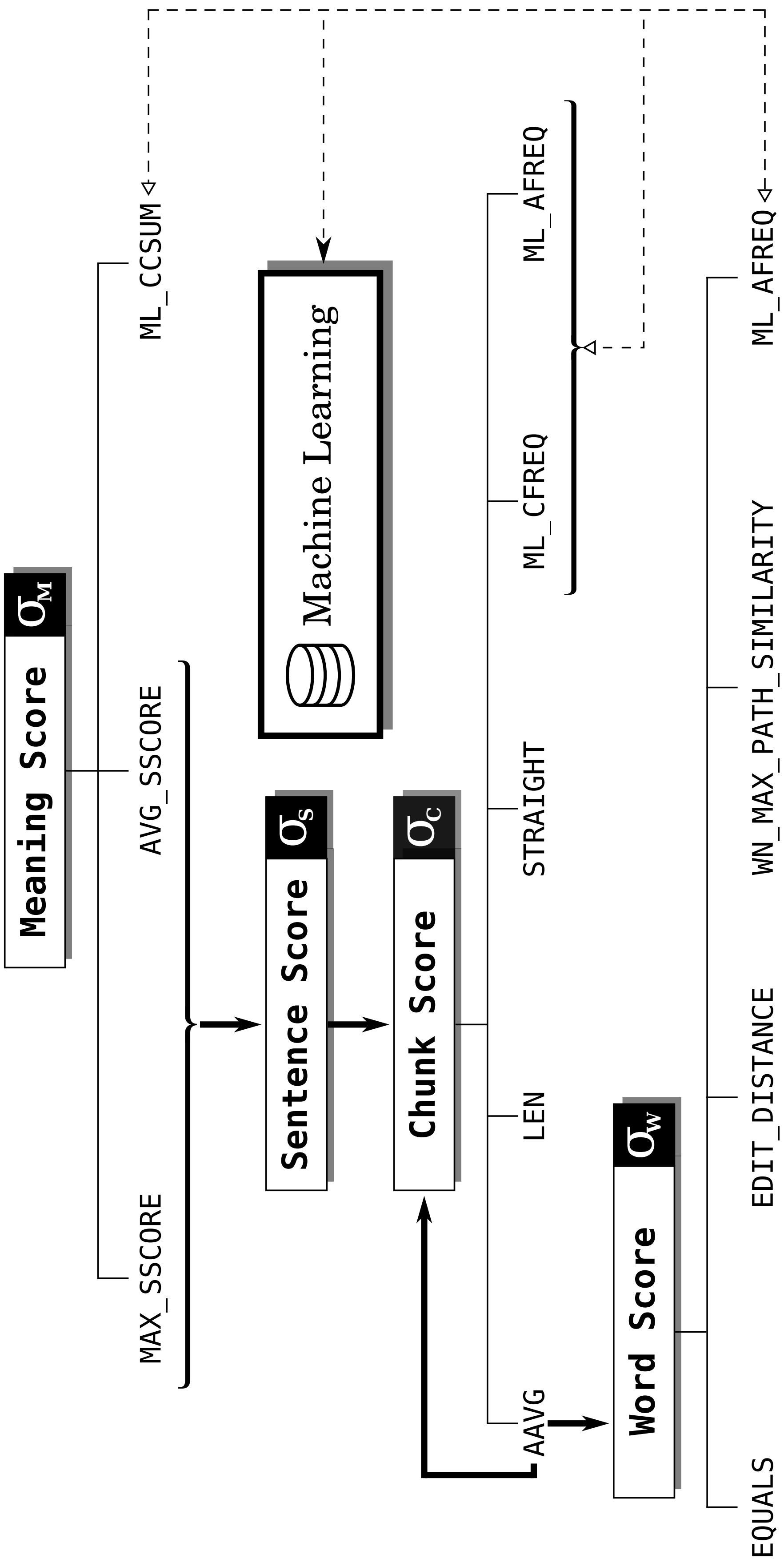

Figure 3.1: Scores' structure

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; 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 similarity score between two words. This score is a linear combination of independent features. At the moment, the following features are taken into account:

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($R$)
```

- Equality (boolean) – 1 if $c_1 = c_2$, 0 otherwise
- Edit distance – $1 - d$, where d is the Levelshtein distance (Levenshtein, 1966) between c_1 and c_2, as it is implemented in NLTK (Loper and Bird, 2002)
- Difference in position – $(1/|p_{c_1} - p_{c_2}|)^{\alpha}$, where p_{c_i} is the position of c_i in the sentence, and α is a free parameter.
- Wordnet (Miller, 1995) Path Length similarity, as it is implemented in NLTK

The weights of the single features, now static, are meant to be trained through Machine Learning

- `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 score 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.

  The score is again the linear combination of independent features, leaving its implementation open and extensible.

  At the moment the only feature being employed is 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 Possibilities

The basic algorithm described in section 4.1 only deals 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 I address the question whether it is possible, and how, for the algorithm to **learn** from the sentences it is parsing. In other words, given a sentence of which I know the meaning, how can I use this information 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.

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.

4.2.2 Class-conditional chunk weight

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 can 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 can be used 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.

4.3 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, this paper only describes the **backbone** of the algorithm, which is not yet ready for more formal tests on existing corpora. The main steps ahead before expecting good results from it include a better tuning and possible extension of the existing features and the implementation of Machine Learning techniques to include alignment probabilities and more advanced statistical features for the chunks being compared (eg. their degree of informativeness).

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 case asking **pertinent 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 its knowledge about the language.

Section 5.1 describes how the best interaction policy is selected; 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 grounded 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 is not sure about the meaning 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 episode

Once the need for grounding is enstablished (5.1), and the software has detected the core of the uncertainty (5.2), it is necessary to produce a dialogue 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 achieved 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, even though only the simplest part of it is implemented in the dialogue manager, the plan produced by the Language Unit does differenciate the two situations, and includes nested questions, which utterance is conditioned on the answer of the top level ones.

5.3.2 Implementation

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.

Chapter 6

Conclusions

future/open questions

- Multi-pass matching: restricting the match candidates changes the conditional probabilities of chunks (eg. with ”the volume”) you select ”increase” or ”decrease”, then conditional relevance of ” the volume” becomes 0 and you can disambiguate the verb.

- Parameters handling (eg. play all the songs by ¡X:ARTIST¿, where system has a model for the parameter type ”ARTIST”)

- Cache partial scores to make computation faster

→ Update with math instead of recomputing features

→ Degree of confidence above which scores are no more computed

→ → Break and return if perfect match is found

- coarse-to-fine matching (restrict candidates with cosine -¿ M2 on remaining)

- Keep a connectionist-like model of matches across utterances: the matches or words fired with the previous utterances remain active in memory for some time before decaying. A model of decay can also be envisioned, which controls the time before stuff decays (eg. if one utterance is ”nevermind”, the model can force immediate decay of everything)

- Machine Learning on weights

- maybe dynamic weights (eg. disregard other Word features if Equals=1)

- Tabu game to get corpus

- Weighting of sources (use confidence of sentences)

- Better integration with OpenTDM (requires documentation)

- Syntactic compositionality: understanding 2 moves with a sentence

- Meta-learning of episodes (eg. new ways to do disambiguation interactions)

- Insert user's negative feedback on wrongly understood sentences

- Reinforcement propagation through fuzzy synsets

- "Economy" of reinforcements: now "Volume" gets 300 times the mass of other chunks: might be useful to enforce subtraction from somewhere where it is necessary to add somewhere else

- How to deal with holes and plugs in training sentences (eg. "clean X up")

—-

Classification

- Perceptron

- Decision stumps

- Naive Bayes

Bibliography

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`.

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.

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.

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

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`.

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`.