Karlsruhe Institute of Technology

Easily Bootstrappable Statistical Spoken Dialogue System

Diploma Thesis of

Krassimir Valev

Interactive Systems Laboratories
Carnegie Mellon University, Pittsburgh, USA
Karlsruhe Institute of Technology, Germany

Reviewer:	Prof. Alexander Waibel
Second reviewer:	Prof. Florian Metze
Advisor:	Prof. Alexander Waibel
Second advisor:	Dr. Liang-Guo Zhang

Duration:: 21. July 2014 – 20. January 2015

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, January 20th, 2015

. . . .

(**Krassimir Valev**)

Zusammenfassung

Klassische, regelbasierte Dialogsysteme benötigen viele Gestaltungs- und Neugestaltungs-Schritte, um leistungsoptimierte Strategien zu erlernen. Des Weiteren, solche fest programmierte Ansätze verweigern die System-Wiederverwendung für andere Szenarien, sind nicht skalierbar und erfordern wesentliche manuelle Arbeit und technische Kompetenz. Statistische Lernmethoden bieten hingegen deutliche Vorteile. Allerdings in Fällen, wo ein System von Grund auf aufgebaut ist, existieren keine geeigneten Domaindaten, welche ein solches Design ermöglichen. Weiterhin, das Sammeln von Dialog-Daten ohne einen lauffähigen Prototyp ist problematisch.

In dieser Arbeit wurde einen statistischen, nicht datenbetriebenen Prototyp für ein Dialogsystem implementiert und in dem Kontext eines Restaurant-Domaines ausgewertet. Das System nutzt das HIS-Modell, in dem der Systemzustand durch das Benutzerziel, Dialog-Historie und Benutzeraktionen beschrieben wird. Die Grundidee dieses Modells ist ähnliche Benutzerziele in einer Gruppe zusammenzufassen, um die Dimensionen des Zustandraumes zu reduzieren und damit das Erlernen von Dialogstrategien zu ermöglichen. Die Ergebnisse deuten darauf hin, dass ein statistisches, nicht datenbetriebenes Ansatz sowohl handhabbar als auch mächtig ist und kann als Grundbaustein für das Entwickeln von dialogbasierten Sprachsystemen dienen.

Um die Leistung des Systems in einer geräuschvollen Umgebung zu verbessern, wird ein realistisches Konfusionsmodell vorgestellt. Mittels natürlicher Sprache werden von Benutzeräußerungen Sätze erzeugt und anschließend durch ein Ausprachewörterbuch und eine Phonemähnlichkeitszuordnung zu einem Phonem-Graph umgewandelt. Der ergebende Graph wird durch einen Sprachverstehen-Modul verarbeitet, welcher schließlich eine Liste von Konfusionen erzeugt. Eine Vielfalt von Techniken wurden verwendet, um die Spracherkennung und Spracherzeugung bei dem Konfusionsmodell automatisch zu initializieren. Dieses Modell ist konform mit dem Ziel dieser Arbeit und ist szenariounabhängig. Die Experimente zeigen, dass das vorgestellte Modell das Benehmen des Systems in geräuschvoller Umgebung im Vergleich zu anderen, nicht datenbetriebenen Ansätze verbessert. Die Schwäche ist die lange Rechenzeit für längere Sätze; weitere Optimierungen sind für eine Online-Anwendung notwendig.

Abstract

Conventional, rule-based, dialogue managers require many expensive iterations of manual design and re-design in order to produce good strategies. In addition, such hand-coded strategies are not reusable from task to task, are not scalable, and require a substantial amount of human labour and expertise. Statistical learning approaches offer significant advantages over the standard rule-based hand-coding approach to dialogue systems development. However, in cases where a system is designed from scratch, there is often no suitable in-domain data to enable such a design and collecting dialogue data without a working prototype is problematic.

In this work, a statistical, virtual-data driven dialogue system prototype for a restaurant information domain has been built and evaluated. The system is based on the HIS model, where the state consists of the user goal, the dialogue history and the user action. The main aspects of the model are that similar user goals are grouped into partitions, thus allowing for a single belief to be maintained for each partition, and reducing the full state space into a summary space to enable policy learning. The results prove that the statistical, virtual-data driven approach is both tractable and powerful and that it provides a solid foundation for developing spoken dialogue systems.

To improve the system's success rate in a noisy environment, in this work, a realistic confusion model has been proposed. It uses natural language generation to materialize the user's utterance and the sentence in turn is converted into a phoneme graph using a pronunciation dictionary and a phoneme similarity mapping. The resulting graph is then processed by a language understanding component that produces a list of confusions. Various techniques are applied to automatically bootstrap the language understanding and language generation components used in the confusion model. The model is compliant with the overall goal of this work and is scenario independent. The numerous conducted experiments show that the proposed confusion model improves the system's behaviour in noisy environments compared to other non-data driven approaches. The weakness of the proposed confusion model is that it can be computationally expensive in case of long sentences and further optimizations are required for the method to be feasible for online use.

Acknowledgements

I would like to express my sincere gratitude to Prof. Alexander Waibel for reaching out to me, for providing me with the opportunity to participate in the interACT exchange program, and for his constant encouragement and valuable feedback on many aspects of this work. A special thanks to Dr. Liang-Guo Zhang for his competent guidance and continuous support, for his meaningful comments and suggestions which steered me all way through; his dedication to my work is very much appreciated. In a similar vein, I would like to recognize Prof. Florian Metze at Carnegie Mellon University for his helpful hand in arranging for my scholarship at CMU and for his precious advice, which made it a thoughtful and rewarding journey. I would be remiss, if I did not acknowledge my mother, Lubomira Kraleva, for proof-reading the English text.

Last, but not least, I would like to thank my family for their endless love and unwavering support over the years.

Contents

1. Introduction

1.1. Spoken dialogue systems

Spoken dialogue systems (SDS) help people accomplish a task using spoken language. For example, a person might use an SDS to buy a train ticket over the phone, to direct a robot to clean a room, or to control a music player in a car. Building an SDS is a challenging task in large part because automatic speech recognition (ASR) and language understanding are error-prone. Speech recognition accuracy is relatively good for a constrained speech domain, for example recognition of digits, names of places, or short commands, but accuracy degrades as the domain language becomes less constrained. Furthermore, as spoken dialogue systems become more complex, not only do the demands on speech recognition and understanding components increase, but also user behaviour becomes less predictable. Thus, as task complexity increases, on the overall, there is a rapid increase in uncertainty, and principle methods of dealing with this uncertainty are needed in order to make progress in this research area.

One straightforward and well-known approach to dialogue system architecture is to build it as a pipeline of processes, where the system takes a user's utterance as input and generates a system utterance as output. Such a processing chain is shown in Figure 1.1. In this chain, the automatic speech recognizer (ASR) takes a user's utterance and transforms it into a textual hypothesis. The natural language understanding (NLU) component parses the hypothesis and generates a semantic representation of the utterance. This representation is then handled by the dialogue manager (DM), which looks at the dialogue context and generates a response on a semantic level. The natural language generation (NLG) component then generates a textual representation of the utterance, and passes it to a text-to-speech synthesizer (TTS) which generates the audio output to the user.

Automatic speech recognition

The task of the automatic speech recognizer (ASR) is to take an acoustic speech signal and decode it into a sequence of words. The speech signal must first be segmented into utterances that may be decoded. This segmentation is most often done based on the discrimination between speech and silence. When a certain amount of speech, followed by a certain amount of silence is detected, this segment is considered to be an utterance.

Once the speech signal has been segmented, the decoding task takes the acoustic input, treats it as a noisy version of the source sentence, considers all possible word sequences,

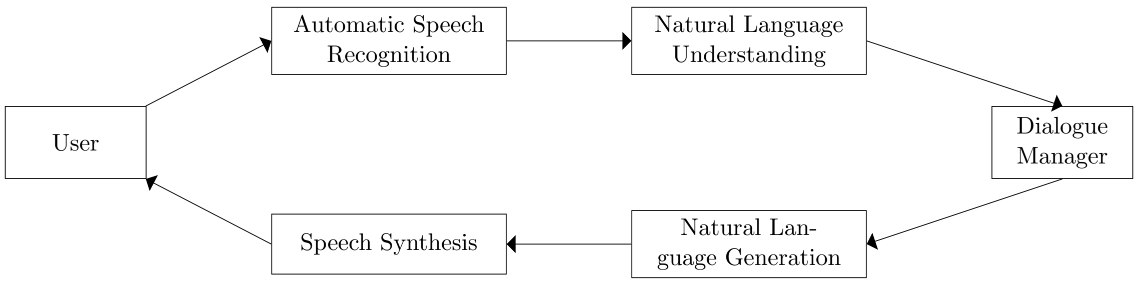

Automatic Speech Recognition
Natural Language Understanding
User
Dialogue Manager
Speech Synthesis
Natural Language Generation

Figure 1.1.: Traditional spoken dialogue system architecture

computes the probability of the sequences generating the noisy input, and chooses the sequences with the maximum probability.

The output from the ASR may consist of a single hypothesis, but an ASR may also return multiple hypotheses, in the form of an n-best list or a word lattice. The hypotheses may also be annotated with confidence scores that contain information about how well the data fit the models.

Natural language understanding

The task of the natural language understanding (NLU) component is to parse the speech recognition result and generate a semantic representation. Such components may be classified according to the parsing technique used and the semantic representations that are generated.

A classic parsing approach is to use a CFG-based grammar which is enriched with semantic information. A more robust approach is to use keyword or keyphrase spotting, where each word or phrase is associated with some semantic construct. The problem with this method is that more complex syntactic constructs (such as negations) are easily lost. Another approach to parse less predictable input is to use a CFG, but extend the parsing algorithm with robustness techniques. The main potential drawbacks with this method are that it may be inefficient, and that it may over-generate solutions that are hard to choose among.

Dialogue management

The dialogue manager is the component that holds the current state of the dialogue and makes decisions about the system's behaviour.

The most common tasks of the dialogue manager can be divided into three groups:

- Input interpretation
- Domain knowledge management
- Action selection

The result of the NLU component is a semantic representation of the user's utterance. However, the NLU component does not have access to the dialogue history, and thus can only make a limited interpretation. For example, in a air travel information scenario when the system queries for a destination and receives a city name as a response from the user, the NLU component can only generate a semantic description which represents the city. Whether the user wants to depart from the city or travel to it has to be inferred by the dialogue manager.

The different knowledge sources needed by a dialogue manager can be separated into dialogue and discourse knowledge, task knowledge, user knowledge and domain knowledge.

The first two are needed for contextual interpretation and action selection. User knowledge can be used to adapt the system's behaviour to the user's experience and preferences. Domain knowledge management includes models (semantic representation of the world) and mechanisms for reasoning about the domain and for accessing external information sources. More complex descriptions of how concepts in the domain model are related are often referred to as ontologies.

The third main task of the dialogue manager is to decide on what the dialogue system should do next, which is often a semantic act that is to be generated and synthesized by the output components. The general idea is to separate an "engine" from a declarative model, in order to make the transition between applications and domains easier.

Central to action selection is that it should somehow be based on the dialogue state. A fairly simple method is to model the dialogue as a set of pre-defined dialogue states in a finite state machine (FSM). The FSM approach is appropriate when the semantics of the dialogue can be simply represented by the current state in the machine. Depending on the number of slots to fill, the FSM can grow to be very large and become incomprehensible for the dialogue designer. To solve this problem, the dialogue state is instead modelled as a store containing variables. Depending on the current state of the store, different actions will be taken. The store does not need to represent the complete dialogue history, but only the parts that are determined to be needed for the application.

An example of a more complex model is the information state approach, where the store is a deep feature structure with variables. The store's control algorithm consists of a set of update rules which have preconditions (on the information state) and effects. Effects may be semantic acts, but also changes to the information state itself, triggering other rules to apply. The structure of the information state, the update rules and the update algorithm may all be customized for the specific application.

In grammar-based approaches to dialogue management, a grammar is used to describe how speech acts can be sequenced and structured. The grammar is used for both input interpretation and action selection. As the dialogue is parsed using the grammar, the resulting tree structure becomes a model of the dialogue history.

Recently, there have been many efforts at making dialogue management (especially action selection) data-driven, by using statistical methods and machine learning approaches. The reasoning behind this effort is that other language technologies have moved in this direction with great success.

Natural language generation

The NLG component takes the semantic representation of the system's utterance and generates a textual representation to be synthesised by a speech synthesizer. The simplest approach to this is so-called canned answers, that is, a simple mapping between a discrete set of semantic acts and their implementations. To make the answers more flexible, templates may be used that contain slots for values.

Research into more advanced and flexible models for NLG has mainly been concerned with the production of written language or coherent texts to be read, dealing with issues such as aggregation and rhetorical structure.

Text-to-speech synthesis

Text-to-speech synthesis (TTS) can generally be divided into two separate problems – the mapping from an orthographic text to a phonetic string with prosodic markup, and the generation of an audio signal from this string. The first problem has been addressed with both knowledge-driven and data-driven approaches. To the second problem, there are three common approaches: formant synthesis, diphone synthesis and unit selection. A formant synthesiser models the characteristics of the acoustic signal, which results in a very flexible model. However, the speech produced is not generally considered very natural-sounding. In diphone synthesis, a limited database of phoneme transitions is recorded, the synthesised speech is concatenated directly from this database and prosodic features are added in a post-processing of the signal. Unit selection is also based on concatenation; however, the chunks are not limited to phoneme transitions. Instead, a large database is collected, and the largest chunks that can be found in the database are concatenated.

1.2. Goals

The biggest disadvantage of the statistical dialogue frameworks is the fact, that they require a large amount of training data in order to learn the optimal policy. Usually data collected for a particular domain is not transferable to another domain and thus cannot be reused. For example training dialogues for an air travel information system cannot be reused in the context of a tourist information application. The goal of this thesis is the implementation of a dialogue framework which is easily bootstrappable and requires absolutely no training data to learn a well-performing policy. The implemented framework will be used as a basis for future research and evaluation in the context of a statistical dialogue manager and thus needs to be properly designed from architectural viewpoint. This can be achieved by specifying clear interfaces and boundaries between the different modules used throughout the system and it will allow for the dialogue manager to be easily extended or configured as part of a future research. Due to the statistical nature of the SDS, additional systems for training the framework need to be implemented and evaluated.

Furthermore a new approach for simulating noise during the training phase has been proposed, that has the advantage over conventional approaches, that it requires no training data and can be used to easily bootstrap the dialogue manager and improve the overall performance of the system by making it more robust to noise.

1.3. Related work

Commercial dialogue systems are typically implemented by charting system prompts along with possible user responses ([PH05]). The system is represented as a graph, sometimes called call flow, where nodes represent prompts or actions to be taken by the system and the edges provide the possible responses. This approach has been the most effective commercially because prompts can be designed to elicit highly restricted user responses. However, the resulting dialogues can become frustrating for users as their choice is severely limited. Further problems arise when speech recognition and understanding errors occur. In some cases, the system might accept information that is in fact incorrect and elaborate error detection and correction schemes are required to deal with this. As a result, commercial spoken dialogue systems are seldom robust to high noise levels and require significant effort and cost to develop and maintain.

Researchers have attempted to overcome these issues in various ways, many of which fall into two categories. The first is to model dialogue as a conversational game with rewards for successful performance. Optimal action choices may then be calculated automatically, reducing maintenance and design costs as well as increasing performance. These models have largely been based on the Markov decision process (MDP), as for example in [LPE00], [Wal00], [SY02], [Pie04] and [LGHS06]. The choice of actions for a given internal system state is known as the policy and it is this policy which MDP models attempt to optimize.

The second research avenue has been to use a statistical approach to model uncertainty in the dialogue. This allows for simpler and more effective methods of dealing with errors. [P$^+$96], [HP99] and [MWP03] all suggest early forms of this approach. These statistical systems view the internal system state as its beliefs about the state of the environment. The true state of the environment is hidden and must be inferred from observations, often using Bayesian networks (BN). The state of the environment is often separated into different components, called concepts, each of which has a set of possible concept values. Sometimes these oncepts are called slots.

More recently, there have been several attempts to use statistical policy learning and statistical models of uncertainty simultaneously. The resulting framework is called the Partially Observable Markov Decision Process (POMDP) ([RPT00], [WY07a], [BPNZ06]).

The use of POMDPs for any practical system is, however, far from straightforward. As with MDPs, dialogue states are complex and hence the full state space of a practical SDS would be intractably large. Thus, the POMDP policy is mapping from regions in n-dimensional belief space to actions. Not surprisingly, these are extremely difficult to construct and whilst exact solution algorithms do exist, they do not scale to problems with more than a few states/actions.

There are two broad approaches to achieving a practical and tractable implementation of a POMDP-based dialogue system. Firstly, the state can be factored into a number of simple discrete components. It then becomes feasible to represent probability distributions over each individual factor. The most obvious examples of these are the so-called slot filling applications where the complete dialogue state is reduced to the state of a small number of slots that require to be filled ([WY07a], [WY07b]). For more complex applications, the assumption of independence between slots can be broadened somewhat by using Bayesian Networks ([P$^+$96], [HP99], [MWP03], [BPNZ06], [TSY08], [TGK$^+$08]). Provided that each slot or network node has only a few dependencies, by using approximate inference, tractable systems can be built and belief estimates with acceptable accuracy can be maintained ([B$^+$06]).

A second approach to approximating a POMDP-based dialogue system is to retain a full and rich state representation but maintain probability estimates only over the most likely states. Conceptually, this approach can be viewed as maintaining a set of dialogue managers executing in parallel where each dialogue manager follows a distinct path. At each dialogue turn, the probability of each dialogue manager representing the true state of the dialogue is computed and the system response is then based on the probability distribution across all dialogue managers. This viewpoint is interesting because it provides a migration path for current dialogue system architectures to evolve into POMDP-based architectures ([HL08]).

The Hidden Information State (HIS) model advocated in [YSWY07] describes a specific implementation of the second approach. The HIS system uses a full state representation in which similar states are grouped into partitions and a single belief is maintained for each partition. The system typically maintains a distribution of up to several hundred partitions corresponding to many thousands of dialogue states.

Independent of the approach, a key issue in a real POMDP-based dialogue system is its ability to be robust to noise. This issue has been addressed by multiple research groups ([LPE00], [SY02], [GHL05], [PD06], [RL06], [SWSY06], [STW$^+$07], [STY07b], [TSW$^+$07], [LL07], [GKM$^+$08]) by the inclusion of a user simulator to bootstrap the POMDP dialogue system. Example approaches include n-gram models ([LPE00]), goal based models ([Pie06]), and conditional random fields ([JLK$^+$09]).

While the development of user simulation tools is an active research area, the error channel is often either excluded altogether ([LPE00], [RL06], [STY07b]) or simulated by generating random errors at a fixed error rate ([HB95], [AWD97], [WAD98]). There are however some more advanced approaches than just converting the semantic representation obtained from the user simulator into text and then generating confusions at a word level. One can then generate confusions according to the past confusions observed for each word ([PR02]), or from fragment-to-fragment confusions ([STY07a]). Further examples include probabilistic phoneme conversion rules ([DMA03]), weighted finite state transducers ([FLAK02]) and linguistically motivated phone confusion models ([PD06], [JLK$^+$09]).

1.4. Thesis structure

The next chapter provides the theoretical background for the main contributions of this work and introduces the relevant concepts of Markov processes, dialogue management, user and error simulation. Chapter 3 describes the proposed phoneme-based confusion model and the issues that it addresses. Chapter 4 documents the implementation details and architecture of the dialogue manager implemented from scratch for this work. The experiments done during the course of this thesis and the evaluation of the techniques introduced previously are described in Chapter 5. The results and findings are then summarized and discussed in Chapter 6, followed by an outlook suggesting possible future research.

2. Background

This chapter provides an introduction to background knowledge and concepts that are relevant to this thesis. First, core concepts such as reinforcement learning and Markov decision processes are explained. The HIS section provides an overview of the dialogue manager architecture, the internal state representation used and a multitude of different concepts and probability models which make up the core of the system. The user simulator section introduces the framework and simulation model used to train the dialogue manager. The chapter concludes by an overview of the error channel framework and its components.

2.1. Reinforcement learning

Reinforcement Learning (RL) is learning through direct experimentation. It does not assume the existence of a teacher that provides training examples. Instead, the learner receives signals (reinforcements) from the learning process, indications about how well it is performing the required task. These signals are usually associated to some condition — e.g., accomplishment of a subtask or complete failure, and the learner's goal is to optimize its behaviour based on some performance metric (usually minimization of a cost function). In order to do this, the learning agent must learn the conditions (associations between observed domain states and chosen actions) that lead to rewards or punishments. In other words, it must learn how to assign rewards to past actions and states by correctly estimating the costs associated to these events. This is different from supervized learning where the rewards are implicitly given beforehand as part of the training procedure.

The accumulation of experience that controls the behaviour (action policy) is represented by a cost function whose parameters are learned as new information is presented to the agent. The agent is also equipped with sensors that define how observations about the external process are made. These observations may be — if necessary — combined with past observations or input to a state estimator, defining an information vector or internal state which represents the agent's belief about the real state of the process. Given the experience obtained so far, the cost function then maps these internal states and presented reinforcements to associated costs. Finally, these costs guide the action policy. The built-in knowledge may affect the behaviour of the agent either directly, altering the action policy or indirectly, influencing the cost estimator or sensors.

The experience accumulation and action taking process is represented by the following sequence. At a certain moment of time, the agent:

1. Makes an observation and perceives any reinforcement signal provided by the process.
2. Takes an action based on the former experience associated with the current observation and reinforcement.
3. Makes a new observation and updates its accumulated experience.

Nearly all RL methods are based on the Temporal Differences (TD) algorithm ([SB98]). The fundamental idea behind it is prediction learning: when the agent receives a reinforcement, it must somehow propagate it backwards in time so that states leading to that condition and formerly visited may be associated with a prediction of future consequences. This is based on an important assumption on the process' dynamics, called Markov condition: the present observation must be conditioned on the immediate past observation and input action. In practical terms, this means that the agent's sensors must be good enough to produce correct and unambiguous observations of the process states.

2.1.1. Markov decision process

Statistical approaches to dialogue management enable extensible dialogue managers to be built based on data rather than hand-coded rules. In particular, the reinforcement learning approach enables a dialogue policy to be learnt in such way as to optimize overall dialogue success. In order to deploy reinforcement learning for policy optimization, dialogue is modelled as a Markov Decision Process (MDP).

An MDP is a more specialized version of a stochastic environment which accounts for the temporal nature of dialogue actions: by taking an action the agent actively changes the environment and influences the states and actions that are available in the consequent dialogue. The Markov Property requires that the state and reward at time $t+1$ depend only on the state and action at time t, as expressed in Equation 2.1.

$$P(s_{t+1}, r_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, s_0, a_0) \approx P(s_{t+1}, r_{t+1} | s_t, a_t) \tag{2.1}$$

An MDP is defined by the tuple $\langle S, A, T, R \rangle$. The state space S refers to the set of reachable states for the agent within the MDP. The action set A contains all actions available to the agent and is often limited to a small number of actions. The state transition function T describes the dynamics of the environment, which next state $s' \in S$ is likely to follow when taking action $a \in A$ in state $s \in S$. Similarly, given a current state s_t and an action a_t, the expected reward value of the next state s_{t+1} is represented by the reward function R.

2.1.2. Partially observable MDP

Formally, a POMDP is defined as a tuple $S_m, A_m, T, R, O, Z, \lambda, b_0$ where S_m is a set of machine states; A_m is a set of actions that the machine may take; T defines a transition probability $P(s'_m | s_m, a_m)$; R defines the expected immediate reward $r(s_m, a_m)$; O is a set of observations; Z defines an observation probability $P(o' | s'_m, a_m)$; λ is a geometric discount factor $0 \leq \lambda \leq 1$; and b_0 is an initial belief state.

At each time step, the machine is in some unobserved state $s_m \in S_m$. Since s_m is not known exactly, a distribution over states is maintained called a belief state such that the probability of being in state s_m given belief state b is $b(s_m)$. Based on the current belief state b, the machine selects an action $a_m \in A_m$, receives a reward $r(s_m, a_m)$, and transitions to a new (unobserved) state s'_m, where s'_m depends only on s_m and a_m. The machine then receives an observation $o' \in O$ which is dependent on s'_m and a_m. Finally, the belief distribution b is updated based on o' and a_m as follows

$$\begin{aligned} b'(s'_m) = P(s'_m|o', a_m, b) &= \frac{P(o'|s'_m, a_m, b)P(s'_m|a_m, b)}{P(o'|a_m, b)} \\ &= \frac{P(o'|s'_m, a_m) \sum_{s_m \in S_m} P(s'_m|a_m, b, s_m)P(s_m|a_m, b)}{P(o'|a_m, b)} \\ &= k \cdot P(o'|s'_m, a_m) \sum_{s_m \in S_m} P(s'_m|a_m, s_m)b(s_m) \end{aligned} \tag{2.2}$$

where $k = \frac{1}{P(o'|a_m,b)}$ is a normalization constant ([KLC98]). Maintaining this belief state as the dialogue evolves is called *belief monitoring.*

At each time step t, the machine receives a reward $r(b_t, a_{m,t})$ based on the current belief state b_t and the selected action $a_{m,t}$. The cumulative, infinite horizon, discounted reward is called *return* and it is given by

$$R = \sum_{t=0}^{\infty} \lambda^t r(b_t, a_{m,t}) = \sum_{t=0}^{\infty} \lambda^t \sum_{s_m \in S_m} b_t(s_m) r(s_m, a_{m,t}) \tag{2.3}$$

Each action $a_{m,t}$ is determined by a policy $\pi(b_t)$ and building a POMDP system involves finding the policy π^* which maximizes the return. Unlike the case of MDPs, the policy is a function of a continuous multi-dimensional variable and hence it can be represented by a set of *policy vectors* where each vector v_i is associated with an action $a(i) \in A_m$ and $v_i(s)$ equals the expected value of taking action $a(i)$ in state s.

The optimal exact value function can be found by working backwards from the terminal state using a process called *value iteration.* At each iteration t, policy vectors are generated for all possible action/observation pairs and their corresponding values are computed in terms of the policy vectors at step $t - 1$. As t increases, the estimated value function converges to the optimal value function from which the optimal policy can be derived. Many superfluous policy vectors are generated during this process, and these can be pruned to limit the total number of vectors. Unfortunately, the pruning itself is computationally expensive, however, approximate solutions can still provide useful policies. The simplest approach is to discretize belief space and then use standard MDP optimization methods ([BS98]). Since belief space is potentially very large, grid points are concentrated in those regions which are likely to be visited ([Bra97], [Bon02]). Each belief point represents the value function at that point and the corresponding optimal action to take is associated with it. When an action is required for an arbitrary belief point b, the nearest belief point is found and its action is used. However, this can lead to errors; hence the distribution of grid points in belief space is very important.

One drawback of grid-based methods is that they do not scale well to large state spaces. The HIS model described below avoids the scaling problem by mapping the full belief space into a reduced summary space where grid-based approximations appear to work reasonably well.

2.2. Hidden Information State Model

2.2.1. HIS POMDP

In the HIS model ([YGK$^+$10]), the dialogue state is represented as a combination of the user goal, the last user act and the dialogue history. This combination can result in a vast number of dialogue states and it would not be computationally tractable to maintain a probability distribution over such a large state space. Therefore, user goals are grouped together into partitions with the assumption that all goals from the same partition are equally probable. Partitions are built using the attribute-value pairs from the N-best list of the user input and the previous system output. They are combined together using the dependencies defined by the domain ontology. The dialogue history is represented by a finite state machine that keeps track of the dialogue progress. The combination of a partition, the associated user act and dialogue history forms a hypothesis, a single point of the partitioned state space. A probability distribution over the most likely hypotheses is maintained during the dialogue and this distribution constitutes the POMDP's belief state.

2.2.2. Domain ontology

As far as dialogue manager is concerned, the dialogue between the system and the user takes place at semantic level, using dialogue acts where each dialogue act comprises a type and a list of attribute-value pairs. For example, *inform(type=restaurant, food=Chinese)* would be the representation at the dialogue act level corresponding to the user saying *"I would like a Chinese restaurant"*. A domain ontology then defines all of the attributes and their possible values, as well as the structural relationship between different attributes.

The ontology has a tree structure. The tree nodes are classified in three groups: class nodes, lexical nodes and atomic nodes. The tree root is a class node and it defines the user goal in the most general way. Other class nodes define the user query more specifically. Class nodes can have many child nodes, the first is always atomic and defines a specific instance of the class, the remainder consist of an optional class node and one or more lexical nodes. Lexical nodes can have only a single atomic child node.

The HIS model makes use of the hierarchical relationship between the attributes to model the dependencies in each user input. For example, in the tourist information domain, if the user specified *food=Italian* this implies that the user wants *type=restaurant* and *entity=venue.*

As previously explained, class and lexical nodes take unique values, for example *type*, *area* or *food.* Atomic nodes, on the other hand, are represented as a set of Boolean indicators for each plausible value from the ontology. Alternatively, an atomic node can be represented as a disjunction of the values which are true or a conjunction of the negation of the values which are false.

2.2.3. Dialogue acts

Dialogue acts in the HIS system take the form *actt*($a_1 = v_1$, $a_2 = v_2$, ...) where *actt* denotes the type of dialogue act and the arguments are act items consisting of attribute–value pairs. Attributes refer to nodes in the user goal state tree described below and values are the atomic values that can be assigned to those nodes. In some cases, the value can be omitted, for example, where the intention is to query the value of an attribute. The same representation is used for both user inputs and the dialogue manager outputs. A full description of the dialogue act set used by the HIS system is given in [You07].

In the HIS system every utterance translates into a single dialogue act. When the speech understanding system is uncertain, the input to the dialogue manager is typically a list of dialogue acts. For example, the utterance "I want an Italian place near the cinema" spoken in a noisy background might yield

inform(type=restaurant,food=Italian, near=cinema) {*0.6*}
inform(type=restaurant,food=Indian, near=cinema) {*0.3*}
inform(type=bar,near=cinema) {*0.1*}

where the number in braces is the probability of each dialogue act hypothesis.

2.2.4. Partitioning

Partitioning is applying a slot value pair $s = v$ to a partition p that contains node s and creating its child partition c. In the ontology, s is either a class or a lexical node and v is an atomic node. In the partition p, node s has a child atomic node that has all possible values that slot s can take. During the partitioning process, the value v in that atomic node of the partition p is set to $false$. The partition c is a copy of the partition p where v is set to $true$. In order to apply slot-value pair $s = v$ for partitioning, it has to be ensured that there is a partition that contains node s. For slot s, the list of superiors is defined as all slot-value pairs $s_i = v_i$ where s_i are class nodes on the path from the node s to the root of the ontology tree, and v_i are the values of their child atomic nodes that enable the attribute expansion leading to the occurrence of s in the tree. The ontology automatically generates this list for each slot s, so that they can be applied prior to applying $s = v$. In this way, it is ensured that there exists a partition with node s before $s = v$ is applied.

The partitioning process starts by applying the list of slot-value pairs form the N-best user input to the initial partition, which is just the root of the ontology tree. The process is then recursively repeated. In such a way, an ordered tree of partitions is created, where the order indicates when each partition was created. Slot-value pairs from the system act are also used for partitioning. It is important to note that this process guarantees that each partition that is created is unique. This is achieved by checking if a partition contains v set to $false$ before $s = v$ is applied to that partition. If it does contain it this means that $s = v$ has already been used and cannot be applied again to that partition.

2.2.5. Probability models

The HIS model uses several probability models to compute the belief of the dialogue hypotheses.

Observation model

The observation model probabilities are derived directly from the N-best list of hypotheses generated by the speech understanding component by assuming that the probability is equal to the posterior probability of the N-best list element corresponding to a_u.

User action model

The HIS user action model is a hybrid model, consisting of a dialogue act type *bigram model* and an *item matching model*

The bigram model reflects the dialogue phenomenon of *adjacency pairs* ([SS73]). For example, a question is typically followed by an answer, an apology - by an apology-downplayer, and a confirmation (”You want Chinese food?”) - by an affirmation (”Yes please.”), or negation (”No, I want Indian.”).

The item matching model is deterministic, assigning either a *full match probability* or a *no match probability*, depending on the outcome of matching the user act with the given user goal partition. For example, the user is not likely to ask about Indian food when the user goal actually indicates that he wants a Chinese restaurant. Therefore, the item arguments of an *inform* act should match the partition. On the other hand, a negation is not likely if the content of the last system act matches the partition. The matching probabilities themselves are optimized empirically.

Dialogue history model

The purpose of the dialogue history model is to track the status of the attributes and values which comprise the user goal, using a grounding-model ([Tra99]). Each terminal node in the associated partition is assigned a grounding state, where these states are updated according to a simple set of transition rules as the dialogue progresses.

The grounding states of nodes in user goal trees are not deterministic, as any node may have multiple possible states depending on the possible dialogue histories that led to the current state. The actual probability returned by the dialogue history model is deterministic. If after updating the history, a resulting hypothesis is inconsistent, for example the user has denied a goal in the partition, then the probability is ≈ 0, otherwise ≈ 1.

2.2.6. Policy representation

As mentioned in Section 2.2, the HIS system represents policies by a set of grid points in *summary belief space* and an associated set of summary actions. Beliefs in *master space* are first mapped into summary space and then mapped into a summary action via a dialogue policy. The resulting summary action is then mapped back into master space and output to the user. This mapping is necessary because accurate belief monitoring requires that the full content of user goals and dialogue acts be maintained, whereas policy optimization requires a more compact space which can be covered by a reasonable number of grid points.

Summary space

In the HIS system, each summary belief point is a vector consisting of the probabilities of the top two hypotheses in master space; two discrete status variables, h-status and p-status, describing the state of the top hypothesis and its associated partition; and the type of the last user dialogue act. The set of possible machine dialogue acts is also compressed in summary space. This is achieved by removing all act items leaving only a reduced set of dialogue act types. When mapping back into master space, the necessary items are inferred by inspecting the top dialogue hypotheses. A dialogue policy can then be represented as a set of belief points in summary space along with the action to take at each point. In order to use such a policy, a distance metric in belief space is required to find the closest grid point to a given arbitrary belief state.

Master–summary space mapping

The process of mapping between master and summary space is illustrated in greater detail in Figure 2.1. The master space is on the left and consists of a set of dialogue hypotheses. On the right of the figure is the summary space represented by a single vector or belief point.

The policy is shown as an irregular grid of these belief points and the figure shows how a system response is generated by mapping the current belief state b into a summary belief state $\hat{b}$, then finding the nearest stored point in the policy $\hat{b}_i$ which in turn yields a summary action $\hat{a}_m^i$. This is then mapped back into master space by a heuristic which assumes that the selected summary action refers to the top hypothesis and constructs the full machine action a_m taking into account the grounding state of all the nodes in the associated partition.

Policy optimization

As explained in Section 2.1.2, representing a POMDP policy by a grid of belief points yields an MDP optimization problem for which many tractable solutions exist. In the HIS model,

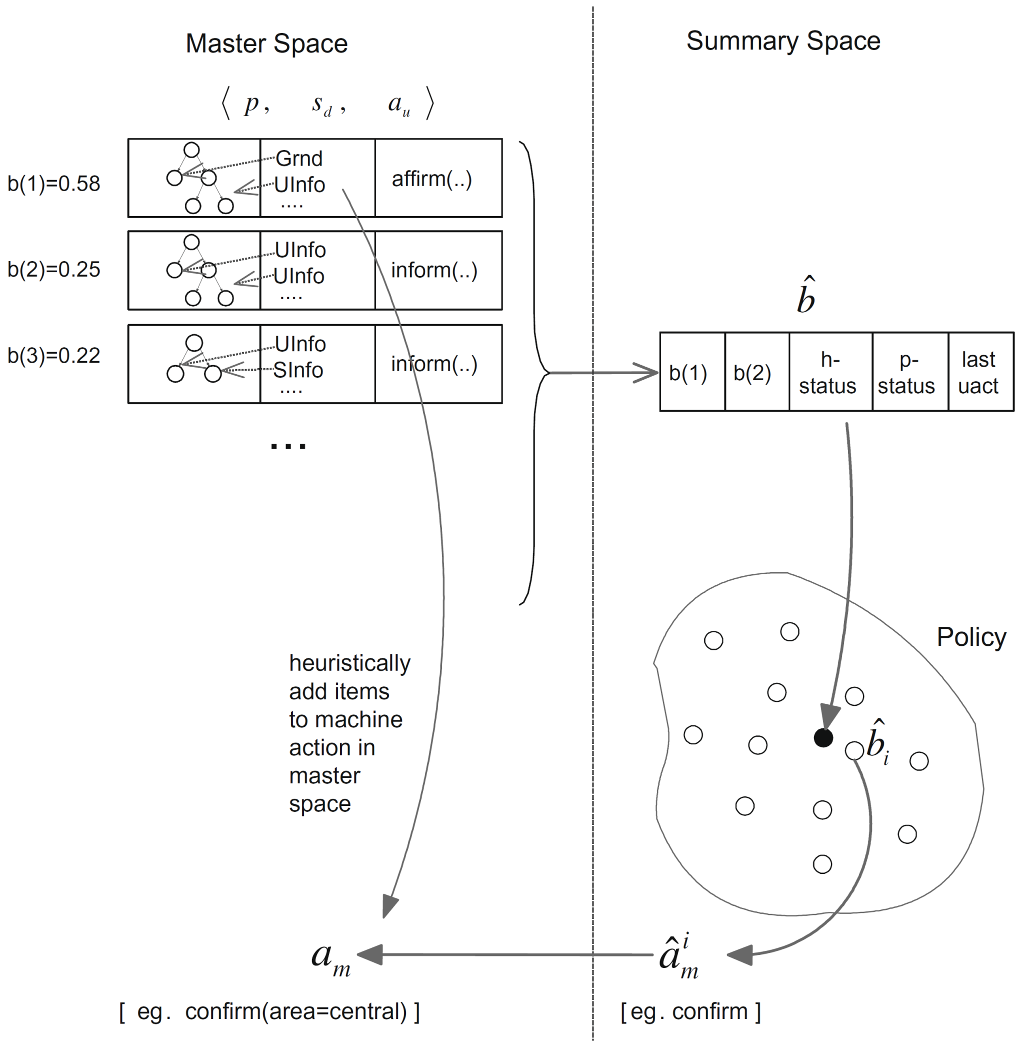


Figure 2.1.: Master-summary state mapping. ([YGK+10])

a simple Monte Carlo Control algorithm is used ([BS98]). Associated with each belief point is a function $Q(\hat{b}, \hat{a}_m)$ which records the expected reward of taking summary action $\hat{a}_m$ when in belief state $\hat{b}$. Q is estimated by repeatedly executing dialogues and recording the sequence of $\langle \hat{b}_t, \hat{a}_{m,t} \rangle$ belief point–action pairs. At the end of each dialogue, each $Q(\hat{b}_t, \hat{a}_{m,t})$ estimate is updated with the actual discounted reward. Dialogues are conducted using the current policy π but to allow previously unvisited regions of the state-action space to be explored, actions are occasionally taken at random with some small probability ϵ.

Belief points are therefore generated on demand during the policy optimization process. Then, every time a belief point is encountered which is sufficiently far from any existing point in the policy grid, it is added to the grid as a new point.

The complete policy optimization algorithm is shown in Figure 1.

k-nn Monte-Carlo Policy Optimization

Alternatively the k nearest neighbour method can be used to obtain a better estimate of the value function, represented by the belief points' Q values. Similarly to the Monte Carlo algorithm, it maintains a set of sample vectors $\hat{b}$ along with their Q value vector $Q(\hat{b}, a)$. When a new belief state $\hat{b}'$ is encountered, its Q values are obtained by looking up its k-nearest neighbours in the state space, then averaging their Q-values.

The complete k-nn version policy optimization algorithm is described in Algorithm 2.

2.2.7. Pruning

Due to the nature of the partitioning process, the number of partitions grows exponentially as the dialogue progresses, which can lead to computational limitations. The complexity issue becomes more apparent if the length of the N-best input is large. Constraining the N-best list to be small and setting a maximum number of dialogue turns can be very limiting for real-world dialogues. Therefore, a pruning technique is needed to deal with the growing number of partitions.

The number of partitions can be reduced simply by removing the low probability partitions. As each partition has a number of associated hypotheses, the probability of a partition is a sum of the probabilities of each of its associated hypotheses. This allows for the low probability partitions to be removed. However, since the partitions represent the groups of user goals, completely removing a user goal makes it impossible to recreate it, which is not desirable.

Partition Recombination

Rather than removing the partitions, the method proposed in [Wil10] reduces the number of partitions by recombining the low probability leaf partitions with their parent partitions. The recombination is performed by removing the complementary value from the parent partition, updating its probability with the probability of its child partition and removing the child partition. An outline of the belief update algorithm that utilizes the partition recombination is given in Figure 3.

This method is shown to be effective in the domains that do not have many slots ([Wil10]). However, there are some considerations in more complex domains. Firstly, it may be limiting to allow a partition to be recombined only with its parent, since there may be other partitions it is complementary to. Secondly, allowing only leaf partitions to be removed might not be desirable in long dialogues, as leaf partitions are usually the last to be created. In dialogues where the user goal evolves with time, the partitions that are created early on become less probable as the dialogue progresses, whereas the leaf partitions are more useful.

Pruning of applied slot-value list

Rather than recombining the partitions, the number of partitions can be reduced by removing some of the applied slot-value pairs ([GY11]). The probability of slot-value pair $s = v$ is the sum of probabilities of all partitions that have v set to *true*. In this way, a sorted list of the applied slot-value pairs can be obtained. The lowest probability slot-value pairs probably have the least impact on the user goal and can be removed.

The partitioning exponentially increases the number of partitions, however, this pruning technique exponentially decreases it, so there is no danger that the number of partition grows faster then being reduced. This allows dialogues of arbitrary length. Furthermore, it also enables large N-best inputs to be applied.

An outline of the belief update algorithm which is used by the pruning method is given in Figure 4. In contrast to the algorithm in Figure 3, the pruning is applied before processing the input, so that no information from the current N-best list is lost before the next system action is chosen.

2.3. User simulator

Online methods for training statistical dialogue managers allow the dialogue policy to be adapted and improved at runtime, i.e. through interaction with real users. During the initial development phase however, many thousand training dialogues are needed to bootstrap the policy, and this is generally too time-consuming and expensive to be done with real users.

The simulation-based approach typically involves two steps. First, a statistical user model (such as an n-gram model) is trained on a limited amount of dialogue data. The model is then used to simulate dialogues with the interactively learning dialogue manager. Simulation is usually done at semantic dialogue act level to avoid having to reproduce the variety of user utterances at word or acoustic level. The simulation-based approach assumes the presence of a corpus of annotated domain dialogues. One of the goals of the user simulator presented here was to be easily bootstrappable without need of any dialogue data for training the user model. Hence, it was necessary to develop a model which was simple enough for the model parameters to be handcrafted and yet capable of producing user behaviour realistic enough for training a prototype system. A similar approach has been previously taken by [LPE00] and [PD06].

2.3.1. Agenda-Based Simulation

Inspired by agenda-based methods to dialogue management, an approach was proposed by [STW$^+$07] that factors the user state into an agenda A and a goal G.

During the course of the dialogue, the goal G ensures that the user behaves in a consistent, goal-oriented manner. G consists of a constraints set C which describe the desired venue, e.g. a centrally located bar serving beer, and a requests set R which contains the desired pieces of information, e.g. the name, address and phone number of the venue.

The user agenda A is a stack structure containing the pending user dialogue acts that are needed to elicit the information specified in the goal. At the start of the dialogue a new goal is randomly generated using the database and the agenda is populated by converting all goal constraints into $inform$ acts and all goal requests into *request* acts. A *bye* act is added at the bottom of the agenda to close the dialogue.

As the dialogue progresses the agenda is dynamically updated and acts are selected from the top of the agenda to form a user act. In response to incoming machine acts, new user acts are pushed onto the agenda and relevant ones are no longer removed. The agenda serves as a convenient way of tracking the progress of the dialogue as well as encoding the relevant dialogue history. User acts can also be temporarily stored when actions of higher priority need to be issued first, hence providing the simulator with a simple model of user memory.

Additionally the user simulator has an initiative model that determines the number of items n that the simulator selects from the agenda stack. In a statistical model the probability distribution over integer values for n should be conditioned on A and learned from dialogue data. For the purposes of bootstrapping the system, n can be assumed independent of A and any distribution $P(n)$ that places the majority of its probability mass on small values of n can be used.

When no restrictions are placed on A and G, the space of possible state transitions is vast. The model parameter set is too large to be handcrafted and even substantial amounts of training data would be insufficient to obtain reliable estimates. It can be assumed that A' is derived from A and that G' is derived from G and that in each case the transition entails only a limited number of well-defined atomic operations. The state transitions can then be decomposed into independent *goal update* and *agenda update* models.

2.3.2. Agenda update model

The agenda transition from A to A' can be viewed as a sequence of push-operations in which dialogue acts are added to the top of the agenda. The agenda update model can be further simplified by assuming that every dialogue act item triggers one push operation. This assumption can be made because it is possible to push a *null*() act (which is later removed) or to push an act with more than one item. In a second "clean-up" step, duplicate dialogue acts, *null*() acts, and unnecessary *request*() acts for already filled goal request slots must be removed.

This model is now simple enough to be handcrafted using heuristics. For example, the model parameters can be set so that when the item $x = y$ in a machine act violates the constraints in G, one of the following is pushed onto A: $negate()$, $inform(x = z)$, $deny(x = y, x = z)$, etc.

2.3.3. Goal update model

The goal update model describes how the user constraints C and requests R change with a given machine action a_m. To restrict the space of transitions from R to R' it can be assumed that each request slot (e.g. address, phone) is either filled using information in a_m or left unchanged. One can further assume that the value of any slot depends on its value at the previous time step, the value provided by a_m and that the transition needs to be conditioned on whether the information given in a_m matches the goal constraints.

To further simplify the model it can be assumed that C' is derived from C by either adding a new constraint, setting an existing constraint slot to a different value (e.g. $food = Thai$), or by simply changing nothing. The transition can be conditioned on simple Boolean flags such as "Does a_m ask for a slot in the constraint set?", "Does a_m signal that no item in the database matches the given constraints?". The model parameter set is then sufficiently small for handcrafted values to be assigned to the model probabilities.

2.4. Error simulator

The error channel can be viewed as a probabilistic model $P(c, \tilde{a}_u|a_u)$, where a_u is the true incoming user dialogue act and $\tilde{a}_u$ is the recognized hypothesis with its associated confidence score c. For the purposes of error simulation, it is convenient to separate the confidence score generation from the error model, as has been previously suggested by [Pie04] and [WY07a].

One promising framework for building spoken dialogue systems is the use of reinforcement learning to learn the optimal decisions to be made. Reinforcement learning algorithms formalize the design criteria of the system as objective reward functions and then optimize the system's decisions to maximize the expected rewards.

When optimizing the rewards, most reinforcement learning algorithms require many more dialogues than are available in the training corpora. An even more significant issue is that most algorithms learn online by interacting with the environment. The standard solution to this is to build a simulation environment, which is used to train the dialogue system ([BS98]). The simulation environment can then be used to generate as many dialogues as necessary.

The simulation environment consists of two main parts. First is the user simulator which, as previously explained, simulates how a user would respond in a given situation. The second component is the error simulator which simulates how the user's response is corrupted. Building systems that are robust to errors is particularly important because both speech recognition and spoken language understanding are prone to mistakes. Previous research has shown that error simulations do have an effect on simulated dialogue performance ([LL07]). Speech recognizers typically output an N-best list of hypotheses along with confidence scores, and so the error simulator should ideally be able to generate similar outputs.

The core component of the error simulator, the *confusion model*, decides what output utterance a given utterance should be confused to. The alternative to generating confusions at the word level is to generate confusions directly at a semantic level. For simplicity, some systems have used a framework where semantic items are either dropped, added, or confused into other items with handcrafted probabilities ([YGK$^+$10]). This is quicker to compute than many word-level approaches but the resulting confusions are unlikely to behave similarly to the real environment. If the number of possible user utterances is limited it is also possible to estimate the confusions using maximum likelihood estimates from a corpus ([WY07a]). This approach becomes difficult when there are many combinations for what the user might say. Once a confused word string is generated, the result can simply be passed through a natural language understanding module to obtain the semantics that the system would have received.

The other component of the error simulator, the *confidence score generator*, determines the length of the N-best list, what the confidence scores are, as well as where in the list the correct hypothesis should occur (if at all). Early dialogue systems were only able to make use of one hypothesis and so there was not much focus on generating full lists of confusions. The focus instead was simply on choosing the confidence score for a single generated confusion. More recently there has been growing interest in partially observable Markov decision processes, which are able to improve performance by using N-best lists of hypotheses ([WY07a], [TYK$^+$10]). These systems directly exploit the extra information in the N-best list in order to provide more robust interaction. For these systems it is particularly important to have good simulations of the N-best list.

When looking into a single confusion, the standard approach is to start by deciding whether the hypothesis should be correct or not, based on a given probability ([WY07a]). The confidence scores are then sampled from two different distributions, one for the correct hypotheses and one for the incorrect. These distributions can be learned from data in various ways, including binning ([STY07a]) and approximation as a sum of exponentials ([PD06]). When taking multiple confusions into consideration, one simple method is to repeatedly generate confusions as for the 1-best case, and then simply assign each confusion a probability proportional to the number of times it appears ([Bli02]). Another option is to generate confidence scores from a parameterized Dirichlet distribution ([TYK$^+$10]).

The input to the error simulator is a user act which is obtained from the user simulator and is a high level semantical representation of the utterance. The system uses the dialogue act set explained prevously, where user acts consist of a dialogue act type followed by a sequence of attribute value pairs.

Given such a user act, the task of the error simulator is to compute a sequence of output acts with associated confidence scores. The number of output acts must also be decided by the error simulator. In order to simplify the process, this generation will be split into several steps. First, the number of output acts is decided with the help of a *confidence score generator*. Given this, a distribution of confidence scores is chosen along with a set of probabilities. Depending on the drawn probability, the correct hypothesis is then either placed at a random position in the list or completely discarded. All other positions are provided with a confused hypothesis along with the confidence score for that position. Confusions are generated by a separate *confusion model*.

Algorithm 1 Monte Carlo policy optimization algorithm.

```
1: procedure TRAIN($reward$)
2:     Let $Q(\hat{b}, \hat{a}_m) \leftarrow$ expected reward on taking action $\hat{a}_m$ from belief point $\hat{b}$
3:     Let $N(\hat{b}, \hat{a}_m) \leftarrow$ number of times action $\hat{a}_m$ is taken from belief point $\hat{b}$
4:     Let $\mathcal{B}$ be a set of grid points in belief space
5:     Let $\pi : \hat{b} \rightarrow \hat{a}_m; \forall \hat{b} \in \mathcal{B}$
6:     repeat
7:         $t \leftarrow 0$
8:         $\hat{a}_{m,0} \leftarrow$ initial greet action
9:         $b = b_0$

           Generate dialogue using $\epsilon$-greedy policy
10:        repeat
11:            $t \leftarrow t + 1$
12:            Get user turn $a_{u,t}$ and update belief state $b$
13:            $\hat{b}_t \leftarrow$ SUMMARYSTATE($b$)
14:            $\hat{a}_{m,t} = \begin{cases} \text{RANDOMACTION()} & \text{with probability}\epsilon \\ \pi(\text{NEAREST}(\hat{b}, \mathcal{B})) & \text{otherwise} \end{cases}$
15:            record $\langle \hat{b}_t, \hat{a}_{m,t} \rangle, T \leftarrow t$
16:        until dialogue terminates with reward $R$ from user simulator
17:    until converged

       Scan dialogue and update $\mathcal{B}$, $Q$ and $N$
18:    for $t \leftarrow T, 0$ do
19:        if $\exists b_k \in \mathcal{B}, |\hat{b}_t - \hat{b}_k| < \delta$ then        ▷ update the nearest point in B
20:            $Q(\hat{b}_k, \hat{a}_m) \leftarrow \frac{Q(\hat{b}_k,\hat{a}_m)*N(\hat{b}_k,\hat{a}_m)+R}{N(\hat{b}_k,\hat{a}_m)+1}$
21:            $N(\hat{b}_k, \hat{a}_m) \leftarrow N(\hat{b}_k, \hat{a}_m) + 1$
22:        else                                                  ▷ create new grid point
23:            add $\hat{b}$ to $\mathcal{B}$
24:            $Q(\hat{b}, \hat{a}_m) \leftarrow R$
25:            $N(\hat{b}, \hat{a}_m) \leftarrow 1$
26:        end if
27:        $R \leftarrow \gamma R$                                    ▷ discount the reward
28:    end for
29: end procedure
```

Algorithm 2 k nearest neighbour Monte Carlo Control algorithm

```
procedure TRAIN($reward$)
    Let $Q(\hat{b}, \hat{a}_m) \leftarrow$ expected reward on taking action $\hat{a}_m$ from belief point $\hat{b}$
    Let $N(\hat{b}, \hat{a}_m) \leftarrow$ number of times action $\hat{a}_m$ is taken from belief point $\hat{b}$
    Let $\mathcal{B}$ be a set of grid points in belief space
    Let $\{\hat{b}_k\}_{knn}$ be a list with the $k$ nearest neighbours of $\hat{b}_t$ in $\mathcal{B}$
    Let t be number of steps it took for the dialog to complete
    for $t \leftarrow T, 0$ do
        if $\exists b_k \in \mathcal{B}, |\hat{b}_t - \hat{b}_k| < \delta$ then        ▷ update the nearest point in B
            for each $\hat{b}_k$ in $\{\hat{b}_k\}_{knn}$ do
                $w \leftarrow \Phi(\hat{b}_t, \hat{b}_k)$        ▷ $\Phi$ weighting function
                $Q(\hat{b}_k, \hat{a}_m) \leftarrow \frac{Q(\hat{b}_k,\hat{a}_m)*N(\hat{b}_k,\hat{a}_m)+R*w}{N(\hat{b}_k,\hat{a}_m)+w}$
                $N(\hat{b}_k, \hat{a}_m) \leftarrow N(\hat{b}_k, \hat{a}_m) + w$
            end for
        else        ▷ create new grid point
            add $\hat{b}$ to $\mathcal{B}$
            $Q(\hat{b}, \hat{a}_m) \leftarrow R$
            $N(\hat{b}, \hat{a}_m) \leftarrow 1$
        end if
        $R \leftarrow \gamma R$        ▷ discount the reward
    end for
end procedure
```

Algorithm 3 Belief Update with Recombination

```
procedure PRUNE($reward$)
    Let $o'$ be an observation from the $N$-best input
    Let $p$ be a partition and its belief $b(p)$
    Let $h$ be a hypothesis and its belief $b(h)$
    repeat  for each dialogue turn
        Belief Update
        Partition each $p$ using slot-value pairs from the last system action $a_m$
        Initialise $b'(p) = 0$ for all partitions $p$ in the current set of partitions
        for each $o'$ in the $N$-best list do
            Partition each $p$ using slot-value pairs from $o'$
            for each partition $p'$ in the current set of partitions do
                for create new hypothesis $h'$ from previous hypothesis $h$ and $o'$ do
                    $b'(h') = P(o'|a'_u)P(a'_u|p', a_m)P(h'|h, p', a_u, a_m)P(p'|p)b(h)$
                    $b'(p') = b'(p') + b'(h')$
                end for
            end for
            Partition Recombination
            Recombine partitions w.r.t the current updated belief $b'(p')$
        end for
        Action Selection
        Choose the next system action $a'_m$ according to $b'(h')$
    until dialogue ended
end procedure
```

Algorithm 4 Belief Update with Pruning

```
 1: procedure PRUNE($reward$)
 2:     Let $o'$ be an observation from the $N$-best input
 3:     Let $p$ be a partition and its belief $b(p)$
 4:     Let $h$ be a hypothesis and its belief $b(h)$
 5:     Let $d$ be a slot-value pair and $p(d)$ its marginal probability
 6:     repeat  for each dialogue turn
            Pruning
 7:         for each applied slot-value pair $d$ do
 8:             $p(d) = \sum_{p:d \in p} \sum_{h \in p} b(h)$
 9:         end for
10:         Prune the list of the applied slot-value pairs w.r.t. $p(d)$
            Belief Update
11:         Partition each $p$ using slot-value pairs from the last system action $a_m$
12:         for each $o'$ in the $N$-best list do
13:             Partition each $p$ using slot-value pairs from $o'$
14:             for each partition $p'$ in the current set of partitions do
15:                 for create new hypothesis $h'$ from previous hypothesis $h$ and $o'$ do
16:                     $b'(h') = P(o'|a'_u)P(a'_u|p', a_m)P(h'|h, p', a_u, a_m)P(p'|p)b(h)$
17:                     $b'(p') = b'(p') + b'(h')$
18:                 end for
19:             end for
20:         end for
            Action Selection
21:         Choose the next system action $a'_m$ according to $b'(h')$
22:     until dialogue ended
23: end procedure
```

3. Phoneme-based error channel confusion model

First, some background information on confusion models and the related work on phoneme-level approaches, as well as on language understanding and language generation, needed for the respective models used during confusion generation, is introduced. Then, the proposed approach to the phoneme based confusion model is presented.

3.1. Introduction

The strength of the stochastic approach when developing a dialogue manager lies in the fact that it naturally takes advantage of large amounts of data. Data is used to estimate model parameters in order to optimize the performance of the system. Usually the more data the system is trained with, the higher the resulting performance on new data is. Moreover, since the training is done automatically and requires little or no supervision, new applications can be developed at the cost of collecting new data, if needed at all. This is both an advantage and a disadvantage of the data driven approach - data collection itself is very expensive and can be really hard to set up. One of the biggest issues when working on a new domain, even with an already implemented dialogue manager is the necessity for training data to bootstrap the dialogue manager. A lot of effort needs to be invested for both handcrafted and statistical dialogue managers in order to train the framework for the new domain. In the case of a handcrafted dialogue system a domain expert needs to model the dialogue flow, all possible user requests and system responses, which involves a lot of manual work. In the case of a statistical spoken dialogue system, as the one described in the previous chapter the effort for training the system is somewhat automated with the user simulator described in Section 2.3, but one still needs to collect training data for the error channel simulator.

Most error simulation approaches do not take into account the acoustic confusability of individual words and utterances. This limitation might be overcome by error simulation based on phonetic confusions ([DMA03], [Pie04], [FLAK02], [SWY04]) where word sequences are first mapped to phoneme sequences using a pronunciation dictionary and then, confusions are generated using a set of probabilistic phoneme conversion rules ([DMA03]), or a handcrafted phone confusion matrix ([Pie04]), or weighted finite state transducers ([FLAK02], [SWY04]). Finally, the confused sequence is mapped back to a word sequence using a dictionary and then may be optionally weighted using a language model.

Although phoneme-level confusion models have been proven to produce promising results, they often need a large amount of training data to model context-dependent phoneme confusions which is not in line with the goal of this work.

In the next section a similar, simple approach is presented which would allow for a completely automated bootstrapping of a spoken dialogue system that is more robust to noise and requires no training data. The only prerequisite is an existing domain ontology (see Section 2.2.2). The algorithm is shown in Figure 6 and the subsequent sections describe the different models used throughout.

3.1.1. Natural language generation

There are many application programs in everyday use that automatically generate texts but only few of these programs use linguistic and knowledge-based techniques. Almost all other systems use programs that simply manipulate character strings, in a way that uses little, if any, linguistic knowledge. The approach is also known as the 'template' approach.

One reason for using natural language generation (NLG) is maintainability; template-based generators can be difficult to modify according to changing user needs ([GDK94]). Making even a slight change to the output of a template-based generator may require a large amount of template rewriting; in contrast, such a change may be straightforward to make in a linguistically-based system.

Another advantage of NLG-based systems is that they can produce higher-quality output. Most applied NLG systems have a sentence planning module that handles aggregation, referring-expression generation, sentence formation, and lexicalization ([RML95]). Performing these tasks well can greatly enhance the readability of a text. Sentences that are comprehensible but ungrammatical can be annoying to the user, and it may be expensive (in terms of programming effort) to set up a template system to correctly handle agreement, morphology, punctuation reduction, and other 'low-level' lexical features. It is straightforward, in contrast, for an NLG system to handle such phenomena.

Another advantage of NLG that might be important in some cases is multilingual output. Support for multiple languages can also be achieved with templates to a certain degree; many systems are localized to other languages simply by inserting a new set of format strings. The quality of texts generated by this approach is not high, but this may be acceptable in some circumstances. At the other extreme, multilingual output could also be achieved by building several separate systems, one for each target language. Such a system would be more expensive to build and might prove difficult to maintain.

The job of the NLG module is to translate the DM intended actions a in a sequence of words t. Several methods are generally used, ranging from using recorded spoken utterances or handwritten prompts to automatically generated sentences. Although most systems use recorded prompts or more generally human authored prompts, the possibility of generating more natural sentences thanks to an NLG system is a novel research area in the framework of SDSs ([WRR02]). For this work an already existing framework for natural language generation was used - SimpleNLG ([GR09], [Rei95], [VR08])

SimpleNLG is a Java library that provides interfaces offering direct control over the sentence realization process, that is, over the way phrases are built and combined, inflectional morphological operations, and linearization. It defines a set of lexical and phrasal types corresponding to the major grammatical categories, as well as ways of combining these and setting various feature values.

Constituents in SimpleNLG can be a mixture of canned and non-canned representations. This is useful in applications where certain inputs can be mapped to an output string in a deterministic fashion, while others require a more flexible mapping to outputs depending, for example, on semantic features and context. SimpleNLG tries to meet these needs by providing significant syntactic coverage with the added option of combining canned and non-canned strings. Another aim of the engine is robustness: structures which are incomplete or not well-formed will not result in a crash, but typically will yield infelicitous, though comprehensible, output. A third design criterion was to achieve a clear separation between morphological and syntactic operations. The lexical component of the library, which includes a wide-coverage morphological generator, is distinct from the syntactic component. This makes it useful for applications which do not require complex syntactic operations, but which need output strings to be correctly inflected.

3.1.2. Natural language understanding

The Phoenix parser is designed for development of simple, robust Natural Language interfaces to applications, especially spoken language applications. Because spontaneous speech is often ill formed and because the recognizer will make recognition errors, it is necessary that the parser be robust to errors in recognition and grammar.

The Phoenix parser maps input word strings onto a sequence of semantic frames. A Phoenix frame is a named set of slots, where the slots represent related pieces of information. Each slot has an associated Context-Free Grammar that specifies word string patterns that match the slot. The grammars are compiled into Recursive Transition Networks (RTNs). When filled in by the parser, each slot contains a semantic parse tree for the string of words it spans. The root of the parse tree is the slot name. In a search algorithm very similar to the acoustic match producing a word graph, grammars for slots are matched against a word string to produce a slot graph. The set of active frames defines a set of active slots. Each slot points to the root of an associated Recursive Transition Network. These networks are matched against the input word sequence by a top-down Recursive Transition Network chart parsing algorithm. The parser proceeds left-to-right attempting to match each slot network starting with each word of the input, as shown in Algorithm 5.

Algorithm 5 Net matching algorithm

```
1: procedure MATCH(input)
2:     for each word in input do
3:         for each slot in active slots do
4:             MATCHNET(slot, word)
5:         end for
6:     end for
7: end procedure
```

The function MATCHNET is a recursive function that matches an RTN against a word string beginning at the specified word position. The function produces all matches for the network starting at the word position, and may have several different endpoints. The networks are not designed to parse full sentences, just sequences of words. The Recursive Transition Networks for the slots call other nets in the matching process. Each time a net match is attempted (all nets, not just slots), this is noted in the chart. All matched networks are added to the chart as they are found. Any time a net match is attempted, the chart is first checked to see if the match has been attempted before. When a slot match is found, it is added to the slot graph. Each sequence of slots in the slot graph is a path. The score for the path is the number of words accounted for by the sequence. Words are not skipped in matching a slot, but words can be skipped between the matched slots. The graph growing process prunes poor scoring paths, just as the acoustic search does. The pruning criteria are first, number of words accounted for and second, the degree of fragmentation of the sequence. If two paths cover the same portion of the input and one accounts for more words than the other, the less complete is pruned. If the two paths account for the same number of words, and one uses fewer slots than the other, the one with more slots is pruned. The resulting graph represents all of the sequences found that have a score equal to the best. The sequences of slots represented by the graph are then grouped into frames. This is done simply by assigning frame labels to the slots. Again in this grouping, less fragmented parses are preferred. This means that if two parses each have five slots, and one uses two frames and the other uses three, then the parse using two frames is preferred. The result is a graph of slots, each labelled with one or more frame labels. Each path through the graph, all scoring equally, is a parse. This mechanism naturally produces partial or fragmented parses. The dynamic programming search produces the most complete, least fragmented parse possible, given the grammar and the input.

3.2. Confusion model

The natural generation library used for this approach allows for a relatively easy generation of sentences out of dialogue acts. Each dialogue act type is mapped to a certain sentence feature. For example *request(address, name=Edeka)* acts generate the sentence *What is the address of Edeka?*. Another example is dialogue acts that require the generated sentence to be a question, e.g. *reqmore()* is materialized as *Are there any more?*. This allows for a generic handling of a multitude of spoken dialogue system domains, without any specific hardcoded domain knowledge.

One weakness of this approach is that some more complicated sentences may sound somewhat ungrammatical to the user, as those require very special case handling, depending on the information that is conveyed. For example some *request()* acts may sound more natural, if the determiner *where* was used instead of *what*. Another example would be the handling of sentences that convey a specific type of information - such as temporal. Thus, the existing system can be improved by introducing a grammatical tagging at the ontology level, supplying more information about the domain keywords. Usually such a task can be automated with a part-of-speech tagger (POS tagger), which is marking up a word in a text (corpus) as corresponding to a particular part of speech (such as noun, verb, adjective), based on both its definition, as well as its context — e.g. relationship with adjacent and related words in a phrase, sentence, or paragraph. Unfortunately none of the POS taggers that were evaluated for this task offer the level of granularity that is required - while all of them correctly identify the word *central* as an adjective, none of them provide more information regarding the type of adjective - locational in this case.

For the task of phoneme generation the CMU pronunciation dictionary was used. Besides the word coverage (125000 North American English words), another big advantage is that it is machine readable and allows for automated offline processing. The current phoneme set has 39 phonemes, not counting variations due to lexical stress. This phoneme set is based on the ARPAbet symbol set developed for speech recognition uses and it is shown on Table C.1.

During application startup the dictionary is read and the data is put into a radix tree data structure. The edges of the tree are labelled with the phonemes from the pronunciation alphabet and the nodes contain the words from the dictionary. By looking up the path from a given node to the root of the tree one will get the pronunciation for the given word. The data structure used allows for fast lookup during runtime and reduces the memory footprint of the application. For example, the phoneme representation of the sentence *I* *am looking for a Chinese restaurant.* would look as follows *AY AE M L UH K IH NG F AO R AH CH AY N IY Z R EH S T ER AA N T.*

The next step in the confusion generation model is to create a phoneme graph out of the phoneme sequence. In order to be able to represent all possible mispronunciations within a sentence a similarity mapping between phonemes is needed. The mapping in Table D.2 has been derived from the WorldBet phonetic alphabet and adapted for ARPAbet. Each phoneme in the previously generated phoneme sequence is expanded to a graph layer, based on the similarity mapping. The phoneme graph for the sentence *I am looking for a Chinese restaurant.* would then be represented by the following graph, which is the input model for the next processing stage.

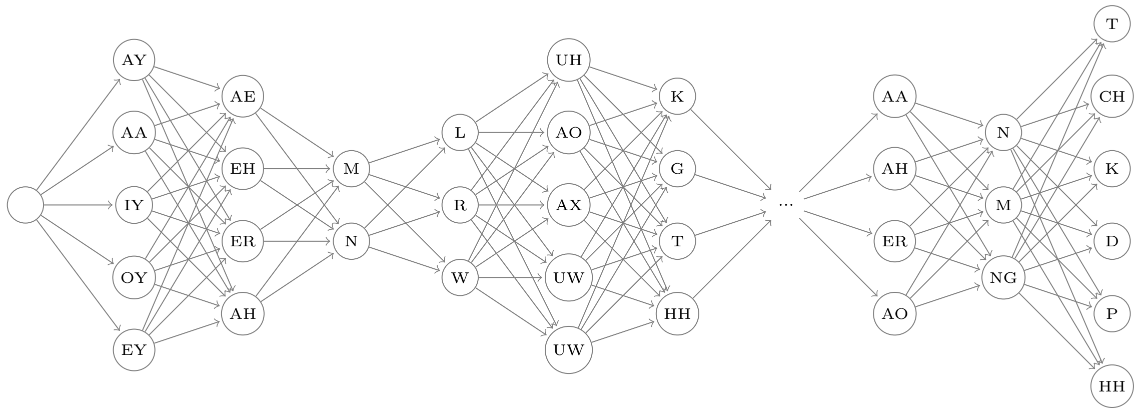


Figure 3.1.: Example phoneme graph

The robustness of the language understanding system depends on how the frames and grammars are structured. Using one frame with one slot will produce a standard CFG parser. This is efficient, but not very robust to unexpected input. At the other extreme, making each content word a separate slot will produce a keyword parser.

In our model we use the latter approach: each content word found in the ontology represents a separate slot. This requires no domain knowledge and provides for easy automation. Once the domain ontology is ready (or at least a release candidate thereof) the system can automatically generate configuration files for the Phoenix parser. These not only contain the aforementioned keywords, but also key-value pairs for all atomic nodes in the ontology. This provides a very easy way to bootstrap the NLU component without any manual work.

One disadvantage of this approach is that while language understanding works quite reliably, the generation of incorrect NLU hypothesis is somewhat limited. This is due to the fact, that the domain ontology usually consist only of few hundred words and within this dictionary there are not many similarly-sounding words. This is why during the grammar generation for the Phoenix parser the domain dictionary has been artificially extended by added synonyms for each word from the WordNet database.

Before feeding the phoneme graph model to the language understanding component all possible graph paths need to be enumerated. Different algorithms for the traversal of the graph were evaluated and the depth-first-search algorithms proved to have the best performance with the data structures currently used throughout the model. Once all confusion candidates are determined they are passed to the language understanding model. The NLU parser then determines if the candidate is a valid sentence by consulting the grammar described previously and returns a semantic representation (see Section 2.2.3) of the sentence. The resulting dialogue acts then represent the confusion model that is used in the error channel framework presented is Section 2.4.

Algorithm 6 Phoneme-based confusion model.

```
 1: procedure CONFUSE($useracts$)
 2:     Let $a_m \leftarrow$ machine dialogue act
 3:     Let confusion list be empty
 4:     $sentence \leftarrow$ GENERATESENTENCE($a_m$)
 5:     $words \leftarrow$ TOKENIZE($sentence$)
 6:     $phonemes \leftarrow$ TRANSCRIBE($words$)
 7:     $graph \leftarrow$ BUILDGRAPH($phonemes$)
 8:     list of confusion candidates $\leftarrow$ TRAVERSEGRAPH($graph$)
 9:     for each $candidate$ in list of confusion candidates do
10:         $confusion \leftarrow$ NLUPARSE($candidate$)
11:         put $confusion$ in confusion list
12:     end for
13:     return confusion list
14: end procedure
```

4. Implementation

One of the most common issues with research frameworks is that they are not very well designed from architectural point of view. Most of them are a collection of scripts, which makes it really hard to incorporate changes in them. The lack of architecture and proper extensibility points were some of the main issues that were addressed in this dialogue manager. The leading design principles were that the framework should be easily extendable, the core objects should be immutable and no domain information should be hardcoded. This is achieved by using two common design principles from software engineering - dependency injection and inversion of control (IoC).

In traditional programming, the flow of the business logic is determined by objects that are statically bound to one another. With inversion of control, the flow depends on the object graph that is built up during program execution. Such a dynamic flow is made possible by object interactions being defined through abstractions. This run-time binding is achieved by mechanisms such as dependency injection or a service locator. In IoC, the code could also be linked statically during compilation, but finding the code to execute by reading its description from external configuration instead of direct reference in the code itself.

Inversion of control carries the strong connotation that the reusable code and the problem-specific code are developed independently even though they operate together in an application. Software frameworks, callbacks, schedulers, event loops and dependency injection are examples of design patterns that follow the inversion of control principle, although the term is most commonly used in the context of object-oriented programming.

Inversion of control serves the following design purposes:

- To decouple the execution of a task from implementation.
- To focus a module on the task it is designed for.
- To free modules from assumptions about how other systems do what they do and instead rely on contracts.
- To prevent side effects when replacing a module.

In dependency injection, a dependent object or module is coupled to the object it needs at run time. Which particular object will satisfy the dependency during program execution typically cannot be known at compile time using static analysis.

In order for the running program to bind objects to one another, the objects must possess compatible interfaces. For example, class A may delegate behaviour to interface I which is implemented by class B; the program instantiates A and B, and then injects B to A.

4.1. Matching

Matching is used during both the partitioning and the belief update processes and is a crucial part of the dialogue manager. The core functionality of the matcher is contained within the *Partition* class (see Figure 4.2). Each dialogue act from the N-best input list is matched against the partition tree. The first step of the matching process is to augment the dialogue act items from the current user act with the act items from the last system act. In case of a *hello()*, *inform()*, *reqalts()* or *request()* only the act items from the user act are considered. In case of an *affirm()* the list with augmented dialogue act items contains all act items from the user act and all act items from the system act, but only if the system act was a *confirm()*.

> sys: confirm(a=x)
> user: affirm(b=y)
> augmented items are *a=x* and *b=y*

If the user negated (*negate()*) the last system act, then in addition to all user act items, negations of all system act items that were not explicitly corrected by the user are considered.

> sys: confirm(a=x)
> user: negate(a=y)
> augmented items are *a=y*

> sys: confirm(a=x)
> user: negate(b=y)
> augmented items are *b=y* and *a!=x*

In case of a *deny()* user act, a negation of the first user act item and all other act items are added to the augmentation list.

> sys: confirm(a=x)
> user: deny(a=y)
> augmented items are *a!=y*

> sys: confirm(a=x,b=z)
> user: deny(a=y,c=w)
> augmented items are *a!=y* and *c=w*

4.2. Partitioning

The implementation of the dialogue manager follows a couple of design principles one of them being the immutability principle. One notable exception is the *Partition* class, which acts as an interface for several different components and thus allows for setting certain properties, such as belief, prior probabilities, partition values and child partitions.

User act	System act	Augmented items
hello(a=x,b=y)	-	a=x, b=y
inform(a=x,b=y)	-	a=x, b=y
reqalts(a=x,b=y)	-	a=x, b=y
request(a,b=x)	-	a, b=x
affirm()	confirm(a=x)	a=x
affirm(b=y)	confirm(a=x)	a=x, b=y
negate()	confirm(a=x)	a!=x
negate(b=y)	confirm(a=x)	a!=x, b=y
deny(a=x,b=y)	-	a!=x, b=y

Table 4.1.: Summary of user and system dialogue act augmentation

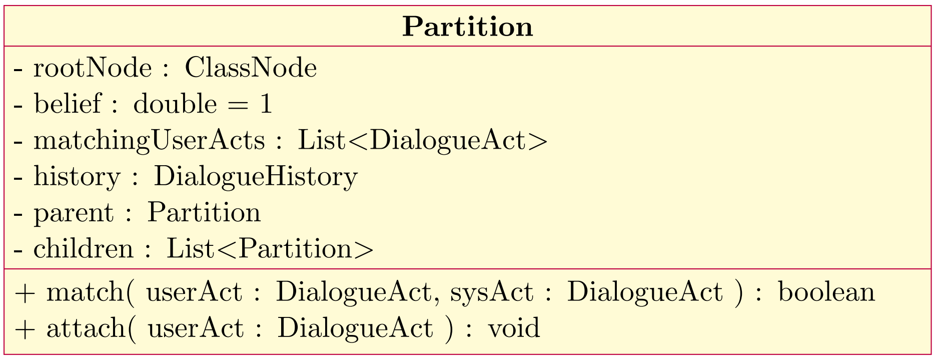

Partition
- rootNode : ClassNode
- belief : double = 1
- matchingUserActs : List<DialogueAct>
- history : DialogueHistory
- parent : Partition
- children : List<Partition>
+ match(userAct : DialogueAct, sysAct : DialogueAct) : boolean
+ attach(userAct : DialogueAct) : void

Figure 4.1.: UML model of the Partition class

4.2.1. User goal state representation

At low level the extended user goal node representation is implemented as a *HashMap* of *String* and *Boolean*. The initial value of the node is empty and the hashmap is filled with values in course of the dialogue. The state representation does not check whether the values for the given lexical in the hashmap are defined in the ontology and no validation is performed. As shown in previous research ([GY11]) incorporating logical expressions as part of the goal state representation has several advantages. This incorporation is achieved with the *Boolean* parameter of the hashmap, which allows for representing multiple values with different signs within a single goal node, e.g. $\neg Chinese \wedge Italian \wedge French \wedge Thai$.

4.2.2. Partitioning

The first step of the partitioning process is to determine whether a partition splitting is even necessary. In case there is already a partition that represents the user's intention, the partitioning step is skipped for the given user act. If the user act cannot be matched against the partition tree, the tree is expanded using the domain ontology rules. First, the dialogue act items from both the user and the last system acts are augmented. Then, for each act item, a superior list is generated by the ontology, which contains all possible paths from the given act item to the ontology root. Given the ontology in Figure A.1, the superior list of the lexical node *name* would contain only one entry (as the node is directly connected to the root ontology node). The list of superiors for *name* would be:

name=null $\rightarrow$ entity=value

A more complicated example would be to query the superiors list of a node, that is connected to the root ontology node via multiple paths. The list of superiors for *hasinternet* would be:

hasinternet=null → type=placetostay → entity=venue
hasinternet=null → drinktype=bar → type=placetodrink → entity=venue
hasinternet=null → drinktype=pub → type=placetodrink → entity=venue

Algorithm 7 Ensure partition algorithm

```
1: procedure EnsurePath($path$)
2:     $current \leftarrow rootpartition$
3:     for each $segment$ in $path$ do
4:         for each $child$ partitions in $current$ do
5:             if $child$ matches $segment$ then
6:                 $current \leftarrow child$
7:                 break
8:             end if
9:         end for
10:        $current \leftarrow$ Partition($current, path$)    ▷ returns the newly created partition
11:    end for
12: end procedure
```

The next step of the partitioning process is to ensure that for every segment of every superior path there is a matching partition in the partition tree. Starting from the root segment in the superior list, the partitioner ensures that a partition containing the given key-value pair exists; if not, the parent partition is split and the required partition is instantiated. The recursive search starts from the root partition and from the root path segment by selecting the root partition as the current partition and the root segment as the current key-value pair. At each step, each child partition of the current partition is queried whether it contains the current key-value pair, and if there is such a child partition, then it is set as the current partition and the process is repeated for the next path segment. If no child partition matches the current path segment, the current partition is split. During the splitting process, the current partition is cloned and the partition slot in the cloned partition is initialized to match the key-value pair (e.g. *entity=venue*). The partition slot is also initialized for the current partition, but the key-value pair is negated (e.g. *entity!=venue*). The newly created partition is then set as the current partition and process is repeated iteratively for every path segment.

Assuming that the user's intent is to find a place with internet connection (expressed with the *inform(hasinternet=true)* dialogue act) and an initial partition tree (with only one node which in turn contains only the root ontology node - *entity()*), the partitioner first has to ensure that there is a partition where *entity=venue* (the root path segment in the superior list). The root node of the partition tree is split so that a new partition is created (*venue()*) and the root partition is modified to include the complement of the current key-value pair (*!venue()*).

When a partition is split the belief for both old and new partition is updated. For the case of a non-terminal node, the partition split probability $P(p\prime|p)$ is specified as a prior in the domain ontology rules (e.g. *entity -> venue(type, area, name, addr, near, phone, comment) 1.0*). This prior can be estimated by counting the occurrences of each class type in a training corpus. However, for the case where an atomic value a is assigned to a terminal node x, using a simple prior for $P(p\prime|p) = P(a|x)$ would severely under-estimate the probability since in practice it will be heavily conditioned by the values of the other terminal nodes in the goal tree. Hence, in this case, $P(p\prime|p)$ is estimated as $ne(x, a, s_u)/ne(x, s_u)$ where the numerator is the number of database entities consistent with the current goal hypothesis s_u when $x = a$ and the denominator is the number of database entities consistent with s_u when x is unspecified.

After calculating the partition priors, the belief for both the old and the new partitions can be determined. To do so first we need to calculate the belief fraction coefficient $bf_{p',p}$, which is based on the partition priors.

$$bf_{p',p} = \begin{cases} \dfrac{P(p')}{(P(p') + P(p))} & P(p') \neq 0 \vee P(p) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Based on the belief fraction coefficient, the belief probability for the newly split partition is $b(p') = b(p)bf_{p',p}$. The belief for the old one is updated to $b(p) = b(p) - b(p')$. The belief fraction is then used to update the belief of the dialogue history hypotheses associated with both partitions. For more information please see Section 4.4.

4.3. Partition pruning

As the dialogue progresses, the number of partitions grows, especially if the environment is noisy and the length of N-best input list of dialogue acts is large, which can cause computational and memory issues. Reducing the number of input items is not a feasible solution, because it can be very limiting for the dialogue manager. The pruning techniques previously described in Section 2.2.7 are made available to the dialogue manager via the interface *IPruningStrategy*.

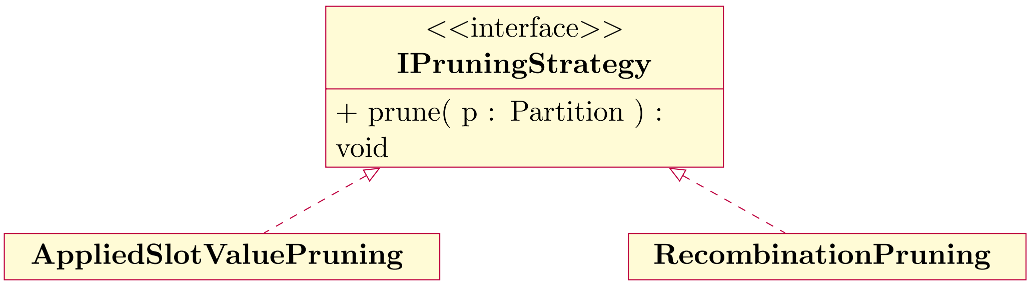


Figure 4.2.: UML model of the *pruning* package

Partition pruning is done at the beginning of the dialogue turn, before processing the incoming N-best list of dialogue acts. The reasoning behind this is to prevent information loss coming from the new dialogue acts. There are few conditions that trigger the pruning process. The first one is that a pruning strategy must have been configured beforehand. This is due to the fact that the *IPruningStrategy* interface (see Figure 4.3) is optional from a dependency injection point of view, so the dialogue manager will boot up and will not complain that some of the required dependencies are missing. The second constraint is that pruning as previously described is only triggered when the number of partitions grows above a certain threshold. This threshold is currently configurable with the configuration property *partitioning.maxNumberOfPartitions*. If both prerequisites are evaluated as *true*, then the root partition from the partition tree is supplied as an argument to the pruning strategy interface.

4.4. Dialogue history

The dialogue history package can be broken down to three classes. Starting from the lowest level, the *SlotHistory* class contains the finite state machine for one user goal node. The state machine itself is represented by a *Stack* structure containing all visited states, transitioning to a new state means pushing the state into the stack. State transitions

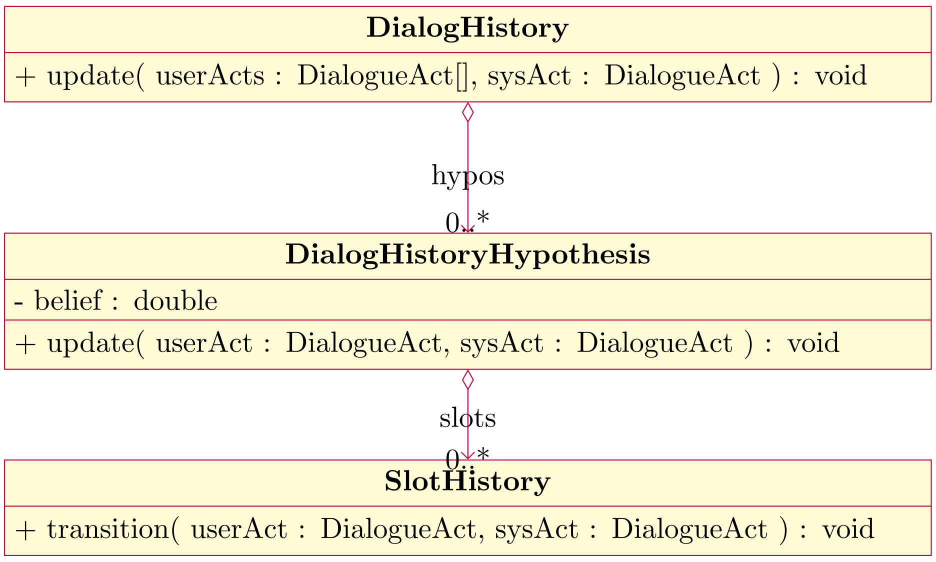

DialogHistory
+ update(userActs : DialogueAct[], sysAct : DialogueAct) : void
hypos
0..*
DialogHistoryHypothesis
- belief : double
+ update(userAct : DialogueAct, sysAct : DialogueAct) : void
slots
0..*
SlotHistory
+ transition(userAct : DialogueAct, sysAct : DialogueAct) : void

Figure 4.3.: UML model of the classes within the *history* package

themselves are basically independent on the current state (with the exception of the final states) so when a state transition takes place, only the type of the input dialogue act is considered.

State	Description
Init	Initial state
UReq	Item requested by user with expectation of an immediate answer
UInf	Item supplied by user during formation of a query
SInf	Item supplied by system
SQry	Item queried for confirmation by system
Deny	Item denied
Grnd	Item grounded

Table 4.2.: List of slot states

The *transition* method assumes the user dialogue act and the last system act as method parameters and then handles the system act transition, user act transition and an optional special case transition in succession. First, the *SlotHistory* checks whether the finite state machine is in a final state (*Deny* or *Grnd*) and if it is, no further transition takes place. Next, the system dialogue act from last turn is handled. If the last system act was a $hello_{system}$ or $inform_{system}$, the state machine transitions to the *SInf* state. Due to a design decision that only the user can ground nodes a transition to the *SInf* state also takes place when encountering a $confirm_{system}$ or $affirm_{system}$ system act. $request_{system}$ and $select_{system}$ cause the state machine to transition to the *SQry* state. Depending on whether the $confreq_{system}$ act has a value for the user goal node (confirming the value to the user) or not (requesting a value from the user), the state machine transitions to *SInf* or *SQry* respectively. The bye_{system} act does not change the grounding state of the node. An exception is thrown, if an unknown dialogue act type is encountered.

Once the system act is handled, the user act is processed in the same way. $hello_{user}$, $inform_{user}$ and $negate_{user}$ transition to *UInf* and $request_{user}$ to *UReq*. The only way that the state machine can enter one of the final states *Grnd* or *Deny* is when the user confirms/affirms or denies the user goal node respectively ($confirm_{user}$, $affirm_{user}$ or $deny_{user}$). No transition takes place when the user closes the dialogue or requests an alternative solution (bye_{user} and $reqalts_{user}$). An exception is thrown, if an unknown dialogue act type is encountered.

The above mentioned special case occurs when the slot name is present in the last system act, but is not explicitly mentioned by the user. In this case, the finite state machine transitions to one of the final states depending on the user act type. The user goal node gets grounded in case of one of the following acts - $hello_{user}$, $inform_{user}$, $confirm_{user}$, $affirm_{user}$, $request_{user}$. If the user does not explicitly mention a slot mentioned by the system, the dialogue manager construes this as an implicit confirmation.

In addition to the *transition* method, the *SlotHistory* offers several public state query-related methods. One can also query the state history using the *getHistory* method.

The dialogue history hypothesis is represented by the *DialogHistoryHypothesis* class which is basically a wrapper around multiple *SlotHistory* objects. The *update* method provides an entry point for updating the dialogue history hypothesis by supplying an user act and the last system act. The method goes on to ensure that there are *SlotHistory* objects for all user goal nodes mentioned in both dialogue acts and then updates them one by one. At the end of the update process, the identical dialogue hypotheses are merged and the belief of the identical hypotheses are aggregated. Two dialogue hypotheses are considered equal, if they apply to the same slots (the slots array for both objects contains the same objects) and if the grounding states for all slots are equal. When looking at the grounding states, only the one at the top of the stack is considered and not the whole history for a given slot. The *DialogHistoryHypothesis* exposes further convenience methods for getting all initialized user goal node names, getting ungrounded slot names, etc.

As previously mentioned, the grounding states of nodes in the user goal trees are not deterministic. Any node may have multiple states depending on the dialogue history that led to the current state. All possible dialogue history hypotheses are encapsulated in the *DialogHistory* class. Within the *update* method, the class takes the n-best list of user acts and the last system act and updates the current hypotheses (and creates new ones, if necessary). If the n-best input list contains multiple dialogue acts, then a snapshot of the current history hypotheses is created and the history itself is deleted. Next, for each dialog act from the n-best list, the snapshot is cloned, each hypothesis in the cloned list is updated and added to the hypotheses list.

The belief probability associated with each *DialogHistoryHypothesis* (see Figure 4.4) is calculated during the partitioning process (see Section 4.2). Using belief fraction coefficient 4.1, the belief for the new and old dialogue history hypothesis is $b(h') = b(h')bf_{p',p}$ and $b(h) = b(h)(1 - bf_{p',p})$, respectively.

4.5. Policy

4.5.1. Handcrafted policy

The handcrafted policy consists of several simple heuristics which determine the summary act to be selected. If the partition is still in its initial state (no partitioning took place), the *greet* action is selected. In case the partition represents a small number of entities (or maybe event uniquely identifies an entity), then the *offer* action is selected. If the top hypothesis is rejected or the user explicitly requested an alternative, the *findalt* action is selected. If the user has requested more information about the selected entity, then *inform* is selected. Finally, if there is an ungrounded user goal node in the top hypothesis, either *confreq* (if the belief is above 0.5) or *confirm* (if the belief is 0.5 or below) is selected; otherwise, a *request* action is selected with the goal of expanding one of the partition's leaf nodes.

Algorithm 8 Update dialogue history

```
procedure UpdateHistory($userActs, sysAct$)
    if Length($userActs$) $\geq 2$ then
        UpdateMultiple($userActs, sysAct$)
    else
        UpdateSingle(First($userActs$), $sysAct$)
    end if
    Cleanup($historyhypotheses$)
end procedure

procedure UpdateSingle($userAct, sysAct$)
    for each $hypo$ in $historyhypotheses$ do
        Update($hypo, userAct, sysAct$)
    end for
end procedure

procedure UpdateMultiple($userActs, sysAct$)
    $snapshot \leftarrow$ HypothesesSnapshot()
    ClearHypotheses()
    for each $userAct$ in $userActs$ do
        $clonedhypotheses \leftarrow$ Clone($snapshot$)
        for each $hypo$ in $clonedhypotheses$ do
            Update($hypo, userAct, sysAct$)
        end for
    end for
    AddToHypotheses($clonedhypotheses$)
end procedure
```

4.5.2. Summary space

Each summary belief point is a vector consisting of the probabilities of the top two hypotheses in master space; two discrete status variables, h-status and p-status, summarizing the state of the top hypothesis and its associated partition; and the type of the last user act.

In case the hypothesis is not yet initialized (this only happens at the start of the dialogue when the partition is still in its initial state and there is no dialogue history) or the partition is still in its initial state, the p-status of the summary belief point is set to *initial.* Otherwise, depending on the number of database entities associated with the partition the p-status is set to *unknown* (no entities), *unique* (one entity), *smallgroup* (three entities or less are associated with the partition) or *hugegroup* (more than three entities).

In case the hypothesis is not yet initialized, the h-status of the summary belief point is set to *initial.* If there are no database entities consistent with the given user goal, the h-status is set to *notfound.* If the user has already accepted an entity that is consistent with the user goal, the h-status is set to *accepted.* If the system has offered an entity that is consistent with the user goal, the h-status is set to *offered.* If the dialogue history associated with the dialogue hypothesis has grounded nodes, the h-status is set to *supported.* Otherwise the h-status is set to *initial.*

Belief points are generated on demand during the policy optimization process. Starting from a single belief point, every time a belief point is encountered which is sufficiently far from any existing point in the policy grid, it is added to the grid. The inventory of

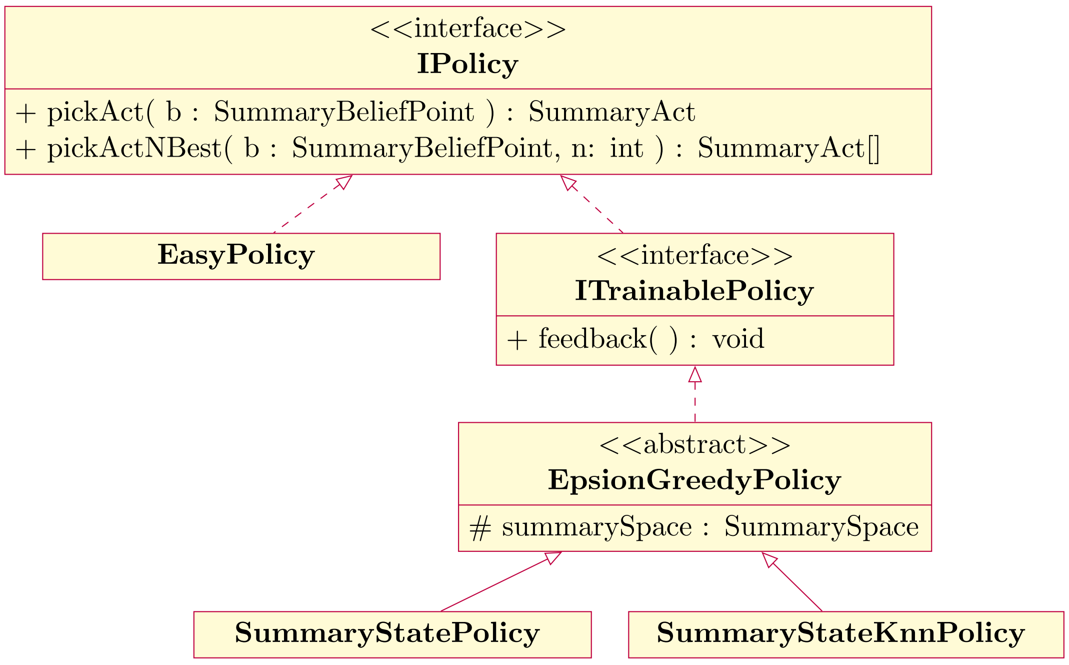


Figure 4.4.: Class hierarchy of the *policy* package.

grid points is thus growing over time until a predefined maximum number of stored belief vectors is reached. The number of maximum belief vectors is currently set to 1000 and can be configured with the *training.maxNumberOfBeliefVectors* property.

4.5.3. Monte Carlo policy

During each dialogue turn, a random number $0 < \beta < 1$ is generated and if $\epsilon > \beta$, a random summary act is chosen. Otherwise, the policy searches for the nearest belief point to $\hat{b}$ in summary space $\mathcal{B}$ and returns the summary action $\hat{a}_m$ that is associated with the highest expected reward. In the edge case that the summary space $\mathcal{B}$ is empty the default *greet* action is selected.

$$\pi(\hat{b}) = \arg\max_{\hat{a}_m} Q(\hat{b}, \hat{a}_m), \quad \forall \hat{b} \in \mathcal{B} \tag{4.2}$$

Additionally, an ϵ value decay is implemented. At the end of each dialogue, the ϵ value is decreased until it reaches a given threshold. The reasoning behind this is to favour exploration of the new states at the beginning of the training epoch and then handle the initiative to the policy as the training progresses. ϵ is set to decrease linearly from 1 to 0.1 within 10000 dialogues, but can be configured with the *policy.epsilonInitial*, *policy.epsilonTreshold* and *policy.epsilonDecreaseRate* properties, respectively.

Algorithm 9 Generate dialog using ϵ-greedy policy

```
1: $\hat{b}_t \leftarrow$ SummaryState($b$)                ▷ summary belief point $b$
2: $\hat{a}_{m,t} = \begin{cases} \text{RandomAction}() & \text{with probability}\epsilon \\ \pi(\text{Nearest}(\hat{b}, \mathcal{B})) & \text{otherwise} \end{cases}$
3: record $\langle \hat{b}_t, \hat{a}_{m,t} \rangle$
```

4.5.4. *k*-nn Monte Carlo policy

An improvement of the Monte Carlo algorithm (presented in [LGJ+09]) which uses the k nearest neighbour method to obtain a better estimate of the value function, represented by the belief points' Q values, was also implemented. To obtain good estimates of the value function interpolation, the k-nn Monte Carlo policy uses a simple weighting scheme based on a nearly Euclidean distance (4.3).

$$|\hat{b}_i - \hat{b}_j| = \sum_{d=1}^{2} \alpha_d \cdot \sqrt{(\hat{b}_i(d) - \hat{b}_j(d))} + \sum_{d=3}^{5} \alpha_d \cdot (1 - \delta(\hat{b}_i(d), \hat{b}_j(d))) \tag{4.3}$$

$$\pi_{knn}(\hat{b}) = \arg\max_{\hat{a}_m} \sum_{\{\hat{b}_k\}_{knn}} Q(\hat{b}_k, \hat{a}_m) \times \Phi(\hat{b}_k, \hat{b}) \tag{4.4}$$

The following kernel function is used for the weighting:

$$\Phi(\hat{b}_1, \hat{b}_2) = \mathrm{e}^{-|\hat{b}_1 - \hat{b}_2|^2} \tag{4.5}$$

4.6. Policy training

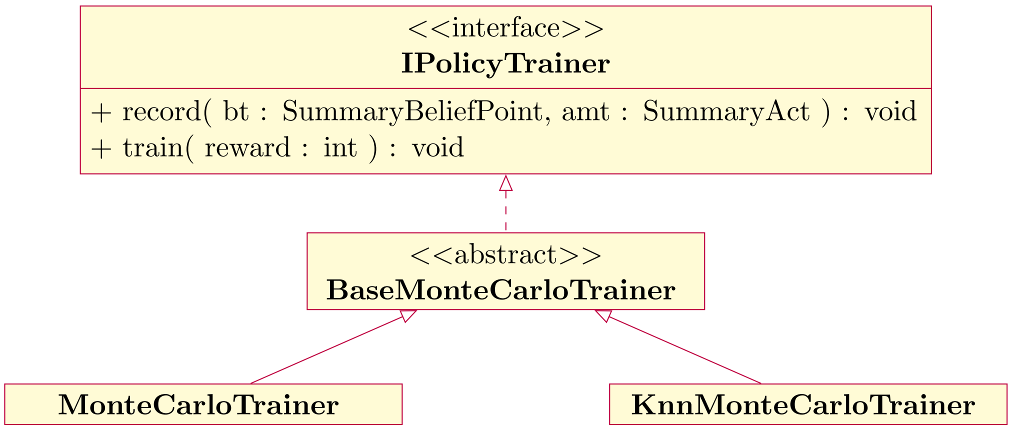


Figure 4.5.: UML model of the major policy training classes

4.6.1. Monte Carlo policy training

The *IPolicyTrainer* interface provides two methods - *record* to record the current belief point $\hat{b}$ and *train* to run the policy training after the dialogue is completed.

During the dialogue the trainer records all belief points $\hat{b}_t$ that were encountered and the summary act $\hat{a}_{m,t}$ that was chosen by the policy. After the dialogue terminates with reward r from the user simulator for each recorded $\langle \hat{b}_t, \hat{a}_{m,t} \rangle$ pair, starting from the last one, the algorithm finds the nearest belief point $\hat{b}_k$ to $\hat{b}_t$ and if the distance between $\hat{b}_k$ and $\hat{b}_t$ is smaller than the threshold δ, the Q and N values of $\hat{b}_k$ are updated (4.6 and 4.7). If the distance exceeds the threshold, the belief point $\hat{b}_t$ is added to the belief grid space $\mathcal{B}$ and the Q and N values are initialized to r and 1, respectively. Finally the reward r is discounted using the discount factor γ (currently set to 0.95 and configurable via *training.discountFactor*).

$$Q(\hat{b}_k, \hat{a}_m) = \frac{Q(\hat{b}_k, \hat{a}_m) * N(\hat{b}_k, \hat{a}_m) + R}{N(\hat{b}_k, \hat{a}_m) + 1} \tag{4.6}$$

$$N(\hat{b}_k, \hat{a}_m) = N(\hat{b}_k, \hat{a}_m) + 1 \tag{4.7}$$

Algorithm 10 Monte Carlo Control algorithm

```
1: procedure TRAIN($reward$)
2:     Let $Q(\hat{b}, \hat{a}_m) \leftarrow$ expected reward on taking action $\hat{a}_m$ from belief point $\hat{b}$
3:     Let $N(\hat{b}, \hat{a}_m) \leftarrow$ number of times action $\hat{a}_m$ is taken from belief point $\hat{b}$
4:     Let $\mathcal{B}$ be a set of grid points in belief space
5:     Let $t$ be number of steps it took for the dialog to complete
6:     for $t \leftarrow T, 0$ do
7:         if $\exists b_k \in \mathcal{B}, |\hat{b}_t - \hat{b}_k| < \delta$ then        ▷ update the nearest point in B
8:             $Q(\hat{b}_k, \hat{a}_m) \leftarrow \frac{Q(\hat{b}_k,\hat{a}_m)*N(\hat{b}_k,\hat{a}_m)+R}{N(\hat{b}_k,\hat{a}_m)+1}$
9:             $N(\hat{b}_k, \hat{a}_m) \leftarrow N(\hat{b}_k, \hat{a}_m) + 1$
10:        else        ▷ create new grid point
11:            add $\hat{b}$ to $\mathcal{B}$
12:            $Q(\hat{b}, \hat{a}_m) \leftarrow R$
13:            $N(\hat{b}, \hat{a}_m) \leftarrow 1$
14:        end if
15:        $R \leftarrow \gamma R$        ▷ discount the reward
16:    end for
17: end procedure
```

4.6.2. *k*-nn Monte Carlo policy training

The k nearest neighbour method can be used to obtain a better estimate of the value function represented by the belief points' Q values. It is very similar to the Monte Carlo Control algorithm but for two minor changes. During the record phase, in addition to the belief point $\hat{b}_t$ and summary act $\hat{a}_{m,t}$ a list of the k nearest neighbours $\{\hat{b}_k\}_{knn}$ for belief point $\hat{b}_t$ is recorded. During the training phase, if there is a belief grid point $\hat{b}_i$ within the distance threshold from $\hat{b}_t$, the Q value is updated by averaging the Q-values of the k-nn nearest neighbours. To obtain good estimates for the value function, local weights are used based on the belief point distance (eq. 4.5).

$$Q(\hat{b}_k, \hat{a}_m) = \frac{Q(\hat{b}_k, \hat{a}_m) * N(\hat{b}_k, \hat{a}_m) + R * w}{N(\hat{b}_k, \hat{a}_m) + w} \tag{4.8}$$

$$N(\hat{b}_k, \hat{a}_m) = N(\hat{b}_k, \hat{a}_m) + w \tag{4.9}$$

4.7. Knowledge base

The *IKnowledgeRepository* interface serves as a repository interface to the entity database. It offers methods to fetch a random entity given some required attribute values (used during the training phase) and queries the system for entities based on some search criteria (used during the interaction between the dialogue manager and the user - either real or simulated).

The entity database to be used for the tourist information scenario can be generated automatically with the provided openstreetmap tool. The starting point is an OSM export (in XML format) containing all desired entities. The XML file can be obtained by visiting http://www.openstreetmap.org/ and lookup the desired city, click on the export button, and write down the bounding box coordinates. Then use the Overpass XAPI to get all amenities (including the metadata) within this bounding box. The order of the bbox coordinates is as follows: west, north, east, south (clockwise starting from west). The openstreetmap tool then reads the exported amenities file and generates a database with

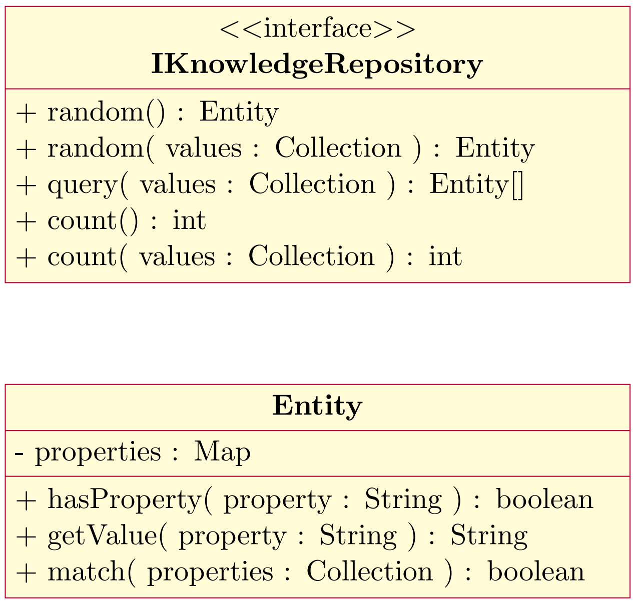

<<interface>>
IKnowledgeRepository
+ random() : Entity
+ random(values : Collection) : Entity
+ query(values : Collection) : Entity[]
+ count() : int
+ count(values : Collection) : int
Entity
- properties : Map
+ hasProperty(property : String) : boolean
+ getValue(property : String) : String
+ match(properties : Collection) : boolean

Figure 4.6.: Class hierarchy of the *data* package.

all valid entities by automatically filling some additional properties, such as nearby city squares or plazas. Entities are considered invalid if the address information is missing and cannot be retrieved via reverse geocoding.

4.8. Dialogue manager

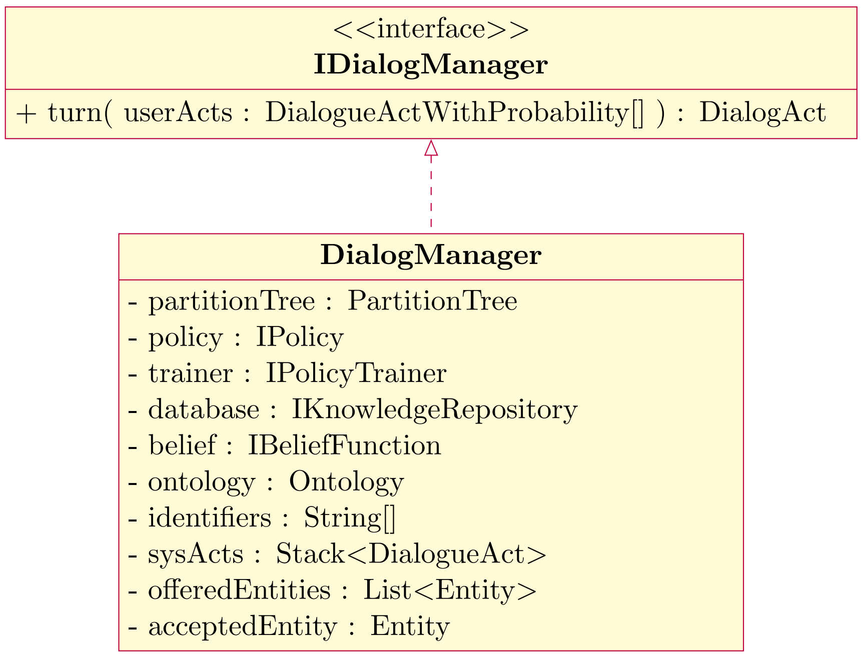

<<interface>>
IDialogManager
+ turn(userActs : DialogueActWithProbability[]) : DialogAct
DialogManager
- partitionTree : PartitionTree
- policy : IPolicy
- trainer : IPolicyTrainer
- database : IKnowledgeRepository
- belief : IBeliefFunction
- ontology : Ontology
- identifiers : String[]
- sysActs : Stack<DialogueAct>
- offeredEntities : List<Entity>
- acceptedEntity : Entity

Figure 4.7.: UML model of the dialogue manager class

The *DialogManager* class is the entry point of the dialogue manager and it binds all of the services explained in this section (see 4.8). The user calls the *turn()* method repeatedly until the dialogue terminates. The pseudocode of the *turn* method is presented in Figure 11.

Algorithm 11 Dialogue manager turn

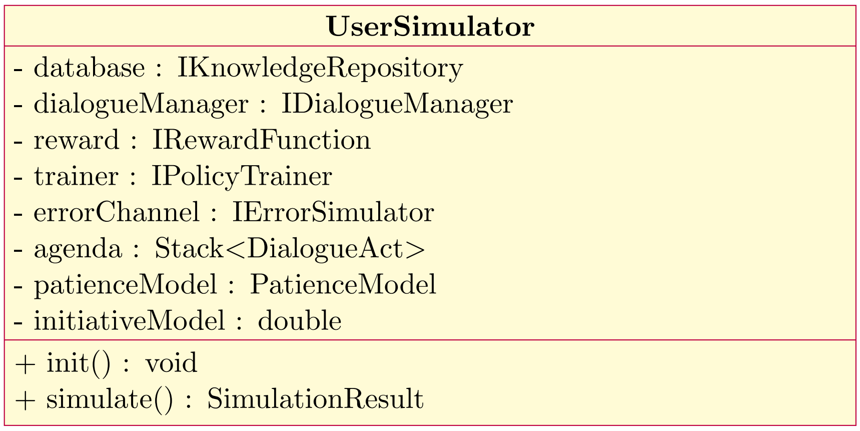
```
 1: procedure TURN($useracts$)
 2:     PRUNE($partitiontree$)
 3:     DETACHALL($partitiontree$)                ▷ detach all previously attached user acts
 4:     for each $useract$ in $useracts$ do
 5:         if $not$MATCH($partitiontree, useract$) then
 6:             PARTITION($partitiontree, useract, lastsystemact$)
 7:         end if
 8:     end for
 9:     for each $useract$ in $useracts$ do
10:         MATCHANDATTACH($partitiontree, useract, lastsystemact$)
11:     end for
12:     UPDATEHISTORY($partitiontree, lastsystemact$)    ▷ user acts are already attached
13:     $hypotheses \leftarrow$ CREATEHYPOTHESES()
14:     for each $hypo$ in $hypotheses$ do
15:         UPDATEBELIEF($hypo$)
16:     end for
17:     $sbp \leftarrow$ GETSUMMARYBELIEFPOINTFROMHYPOTHESES($hypotheses$)
18:     HANDLEOFFEREDENTITIES()
19:     $summaryact \leftarrow$ PICKACT($policy, sbp$)
20:     if $policytrainerispresent$ then
21:         RECORD($trainer, sbp, summaryact$)       ▷ record the summary belief point and
    the summary act
22:     end if
23:     $systemact \leftarrow$ EXPAND($summaryact, hypotheses$)    ▷ use heuristics to expand the
    summary
24:     return $systemact$
25: end procedure
```

Something that has not been addressed previously is the fact that the dialogue manager keeps track of the entities that were offered to the user and whether or not the user has accepted one of them. If they have, then this entity is used for the summary act expansion.

4.9. User simulator

Statistical data driven dialogue managers always require training to learn the optimal policy for interacting with the user. In the case of a reinforcement learning based system, the training process cannot be done manually, due to the sheer number of training dialogues required. For the purpose of automated training, a user simulator has been implemented to interact with the dialogue manager and emulate the user's behaviour. The user simulator is based on the research of Schatzmann et al. [STW$^+$07]. The *UserSimulator* class uses the same design and architectural principles as the dialogue manager. Figure 4.9 provides a better overview of the interface methods and component dependencies of the user simulator.

The first task when starting the user simulator is to initialize it by generating a random goal to be used during the dialogue manager simulation. The goal is setup by fetching a random entity from the entity database based on which the goal constraints and requests are derived. In order to better mimic a real user, the emulated user also has a patience model and an initiative model, that are also initialized in the $UserSimulator$'s *init* method (see Figure 12).



UserSimulator

- database : IKnowledgeRepository
- dialogueManager : IDialogueManager
- reward : IRewardFunction
- trainer : IPolicyTrainer
- errorChannel : IErrorSimulator
- agenda : Stack<DialogueAct>
- patienceModel : PatienceModel
- initiativeModel : double

+ init() : void
+ simulate() : SimulationResult

Figure 4.8.: UML model of the user simulator class

Algorithm 12 User simulator initialization

```
1: procedure Init
2:     $goal \leftarrow$ RandomGoal()
3:     InitGoalRequests($goal$)
4:     InitGoalConstraints($goal$)
5:     InitPatienceModel()
6:     InitInitiativeModel()
7: end procedure
```

After the user simulator has been set up, the actual simulation takes place. Following the initial response from the dialogue manager, the simulation loop described below is executed. Depending on the response of the dialogue manager, a series of responses is pushed to the top of the agenda. The agenda uses a stack data structure, which allows for natural sorting of the dialogue acts to be communicated to the dialogue manager. The next step in the user simulation process is to clean up the agenda by removing unnecessary dialogue acts, as well as duplicate and *null()* acts. The initiative model initialized during the *init* method determines the number of dialogue acts to pop from the agenda which are augmented into a single dialogue act, if possible (only semantic acts with the same act type can be augmented). The ones that cannot be melted together are returned to the top of the agenda. The resulting dialogue item is then fed to the error channel implementation and the resulting noisy N-best list would be the input for the dialogue manager in the next simulation step. The dialogue terminates when the emulated user receives a *bye()* act or it runs out of patience (based on the aforementioned patience model). Afterwards, the policy trainer is notified that the dialogue has ended and that the policy can be trained (if needed). The key facts for the simulation - the reward, the number of turns and whether or not the dialogue was successful are summarized as the simulation result.

4.10. Error channel

As previous research shows [STY07a], artificially introducing noise during the training process can lead to a more robust and better online performing policy. This can be achieved by introducing an error channel as part of the user simulation. A general overview of the semantic level error channel is shown in Figure 4.10.

The first step in the error channel simulator is to determine the maximum length of the N-best list, which is done using the configuration parameter *simulator.errorChannel.numberOfConfusions*. For every item on the list, a probability score is generated by the confidence score generator

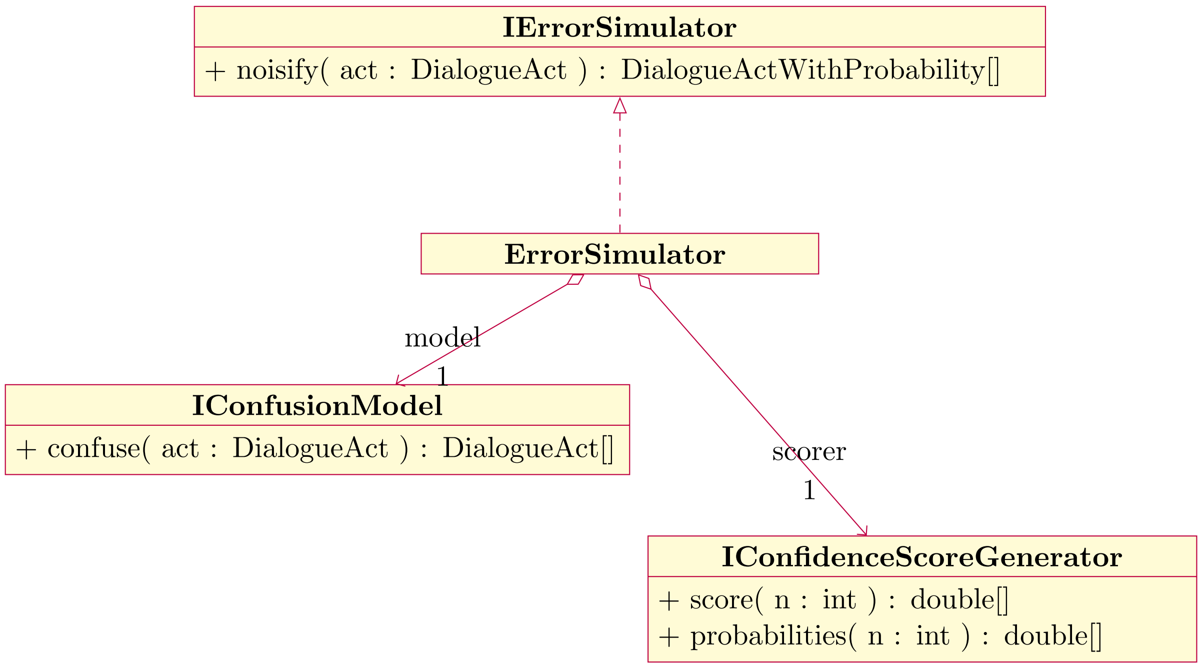

IErrorSimulator
+ noisify(act : DialogueAct) : DialogueActWithProbability[]
ErrorSimulator
model
1
IConfusionModel
+ confuse(act : DialogueAct) : DialogueAct[]
scorer
1
IConfidenceScoreGenerator
+ score(n : int) : double[]
+ probabilities(n : int) : double[]

Figure 4.9.: UML model of the classes within the *error* package

(see Section 2.4). Another probability is drawn that determines the position of the original dialogue act within the N-best list. Each of the other slots in the list can be either the original dialogue act (with the configurable probability *simulator.errorChannel.handcrafted.confusionRate*) or a dialogue act from the confusion model. As this can lead to dialogue item duplicates within the noise list, duplicate items (together with their probabilities) are aggregated before returning the list to the user simulator.

5. Evaluation

The first part of the evaluation will focus on evaluating different system parameters and their influence on the dialogue reward and the number of successfully completed dialogues during training. Later on, the performance of the system will be evaluated and the impact of different confusion models will be compared.

5.1. Experimental setup

The domain for testing the dialogue manager is a restaurant information system with about 35 entries. The goal of the user is to successfully retrieve the telephone number, address and name of a restaurant based on his search criteria - type of cuisine and neighbourhood or nearby place. As previously explained, the dialogue manager requires a large amount of training dialogue data; this has been automated by using a user simulator. Due to the nature of reinforcement learning, around 10 to 15 thousand simulations are required for the dialogue manager to learn a well-performing policy. The training process is not triggered after every simulation, but after a batch of 1000 dialogues; the simulation data is then replayed and the training algorithm updates the values in the Q and N matrices. Both the patience model and the initiative model for the user simulator are enabled per default. The maximum number of partitions has been restricted to 300 and the applied value slot partition pruning algorithm described in Section 2.2.7 is used to reduce the number of partitions, once the configured threshold has been exceeded. The default learning algorithm used for training is the Monte Carlo policy optimization explained in Section 2.2.6 (without the knn term). Furthermore, the following weighting coefficients (α vector) are used for the Euclidean distance function between two summary belief points: the distance between the probabilities of the best hypotheses is weighted by 1, as well as the distance between the probabilities of the second best hypotheses. The distance between the h and p status is weighted by 1.5 and the distance between the last user acts - by 1.25.

5.2. Evaluation

5.2.1. Dialogue manager training

One of the system parameters that has direct influence on the performance of the training policy is the maximum number of vectors in the belief space. The minimum number of belief vectors can be derived from the cardinality of all vector components. Given that the h-status, p-status and dialogue act type space is finite, the minimum number of vectors with respect to those vector elements; there are 6 possible h states as well as p states and 12 dialogue act type resulting in 432 combinations. Furthermore, the two probabilities in the summary space vector can be classified into one of three bins depending on their corresponding values: (1.0, 0.0) - the top hypothesis is certain; (0.5, 0.5) - the top two hypotheses are equally likely; and (0.0, 0.0) - all hypotheses are equally unlikely. Taking this factorization into account when determining the baseline for the maximum number of vectors in the belief space, it was deemed that 1000 vectors should be an appropriate baseline.

Several different values were evaluated and the conclusion is that 750-1000 vectors is the recommended number of belief vectors. As seen from Figures 5.1 and 5.2, selecting only 500 vectors as the upper bound of policy states is quite limiting to the policy's flexibility and has a visible influence on the system's training. Selecting more than 1000 belief states does not bring any advantage in respect to policy performance.

Additionally, the grid-based training can be changed by replacing the Euclidean metric in Eq. 4.3 by a quantizer in which the probabilities of the top two hypotheses are placed in one of aforementioned bins.

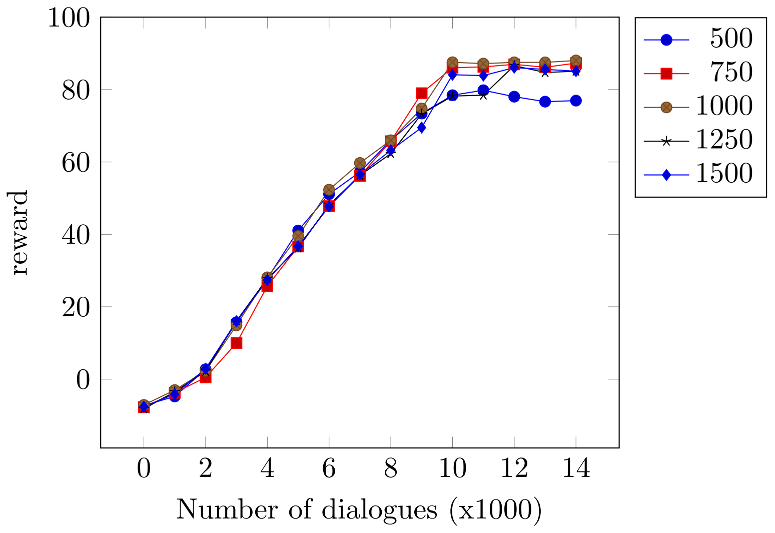


Figure 5.1.: Influence of the number of belief vectors on the dialogue reward.

The next system parameter that was evaluated was the reward for successfully completing a dialogue. Results show that if the positive reward was too small, the training process gets stuck in a local maximum and the training process fails. The selected values that were tested showed that while 20 is insufficient for the dialogue manager to be trained, if a very large value for the reward function is selected, it can also deteriorate the training process. Simulations showed that a value in the 50 to 100 range is optimal (see Figure 5.3).

The goal of the next experiments was to determine the influence of the distance threshold parameter which determines whether a summary belief point will be added to the belief space or a neighbour point will be updated with the reward data from the training. The α weights for the Euclidean function that are described in section 5.1 were not changed during the experiments.

Experiments show that a threshold value of 1 or 1.25 is optimal for the system's performance (see Figures 5.4 and 5.5). A threshold value above 1.5 leads to too great distances between the points in the summary belief space and this reduces the system's precision. The way the distance function and the α weights are setup means that a larger threshold

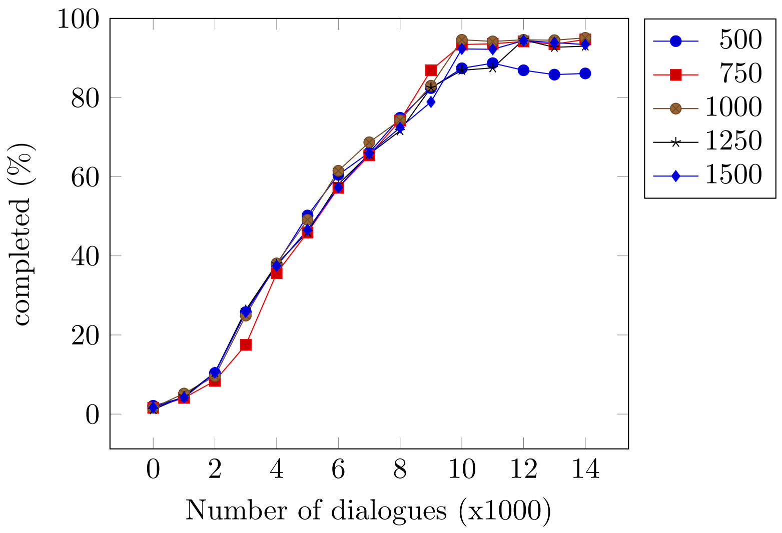


Figure 5.2.: Influence of the number of belief vectors on the number of successfully completed dialogues.

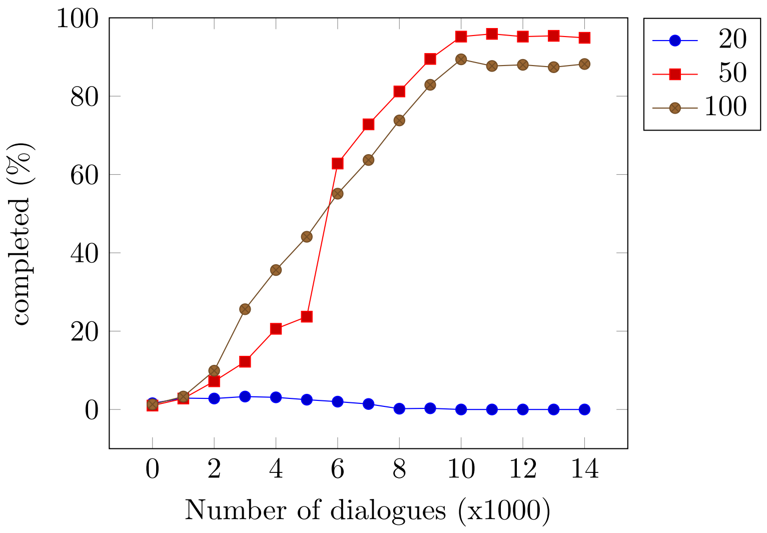


Figure 5.3.: Influence of the reward on the number of successfully completed dialogues.

requires multiple vector point components to be different for the belief point to be added to the belief space.

As previously mentioned, the distance threshold and the weights (α parameter) for the Euclidean distance function are tightly coupled in determining the new summary belief space vectors. The impact of the different weight components will be evaluated next. The experiments were setup as follows - both the distance threshold and the elements of the weights vector were reset to 1; for each experiment the weight for a different component was increased to 1.5 in order to determine the importance of the component for the training process.

Experiments (see Figures 5.6 and 5.7) have shown that the type of the last user dialogue act bears the greatest significance for the policy. Furthermore, the probability of the top hypothesis, as well as the h and p-status of the top dialogue hypothesis prove to influence the performance of the training process. Results show that the learned policy can be quite unstable, if it relies only on the weight of the probability of the second best hypothesis.

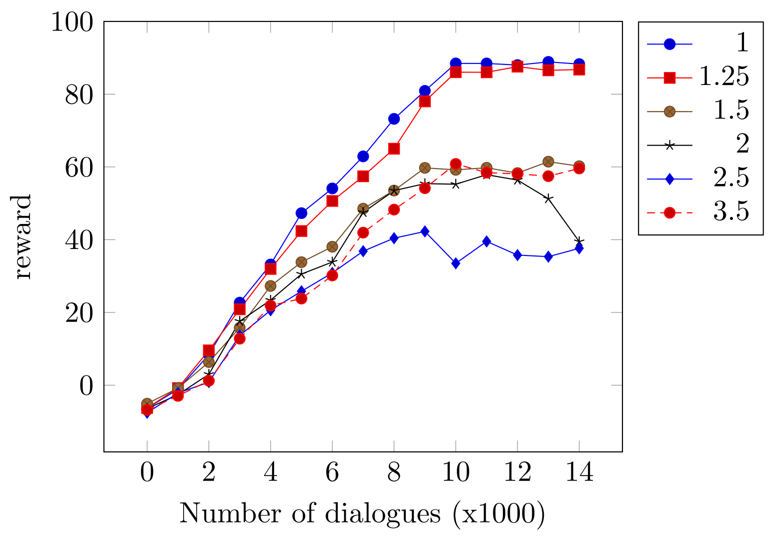


Figure 5.4.: Influence of the euclidean distance threshold on the dialogue reward.

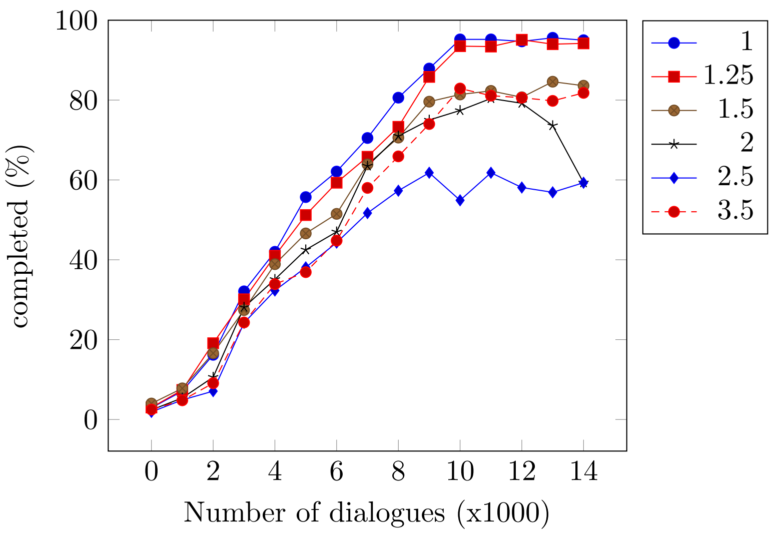


Figure 5.5.: Influence of the Euclidean distance threshold on the number of successfully completed dialogues.

So far, the training setup has used the Monte Carlo training algorithm for the policy training. The next evaluation will compare the performance of the baseline (Monte Carlo with all other parameters set to their default values) with different configurations of the knn-Monte Carlo algorithm. Several k values were evaluated to determine their impact on the dialogue manager performance.

The performance of the baseline Monte Carlo algorithm proves to be very comparable to the performance of the knn Monte Carlo, with the knn parameter set to 3. It appears that there is a linear dependency between the selected knn value and the system's performance. The smoothing applied by the knn Monte Carlo approach seems to be deteriorating by increasing the knn value (as shown in Figure 5.10) and 1 proves to be insufficient for learning a well-performing policy.

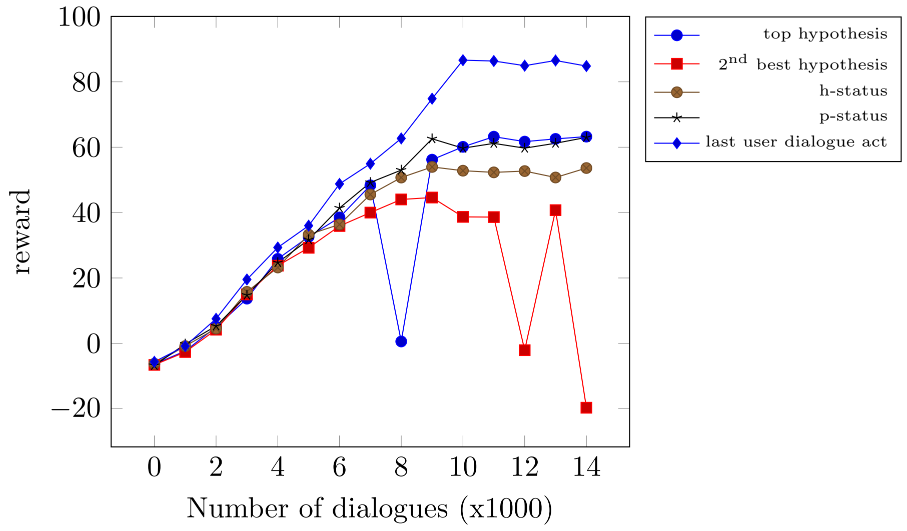


Figure 5.6.: Influence of the Euclidean distance weights on the dialogue reward.

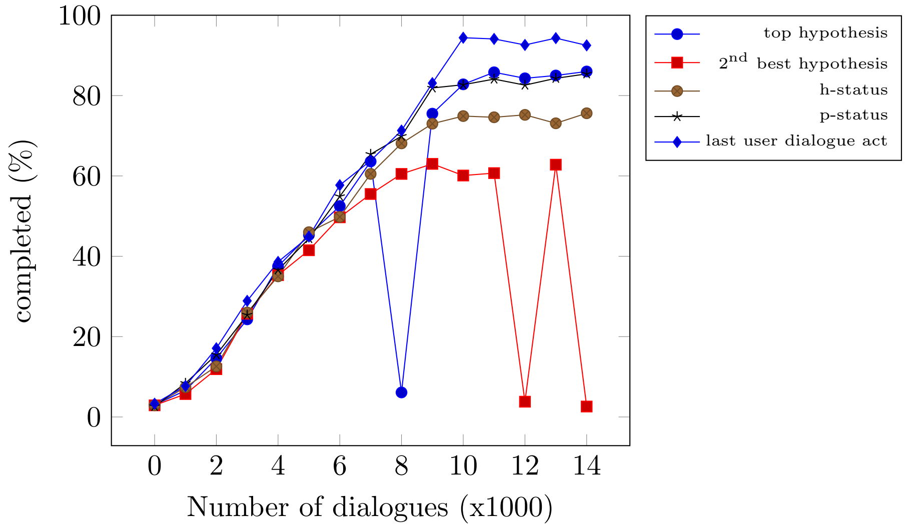


Figure 5.7.: Influence of the Euclidean distance weights on the number of successfully completed dialogues.

5.2.2. Confusion models

After determining the optimal dialogue manager parameters, the next step is to evaluate the impact of the error channel confusion model proposed in Chapter 3 on the system's performance. This approach generates confusions on a phoneme level, initially passing the semantic dialogue act representation through a language understanding, building a phoneme graph, based on similar sounding phonemes and subsequently feeding the graph to language understanding component to determine the valid confusions. In the later experiments it is labelled the *Phoneme* model.

Another simple approach is to iterate through the slot-value pairs of the given dialogue act and confuse each one with a predetermined probability. This approach is used in various past frameworks ([YSWY07], [TYK^{+}10]). When a slot-value pair is confused there is a fixed probability that it will be deleted completely; otherwise, the value is confused uniformly to one of the other values of the slot. There is also a fixed probability that extra items are added or the act type is confused. It will be used here as the baseline approach

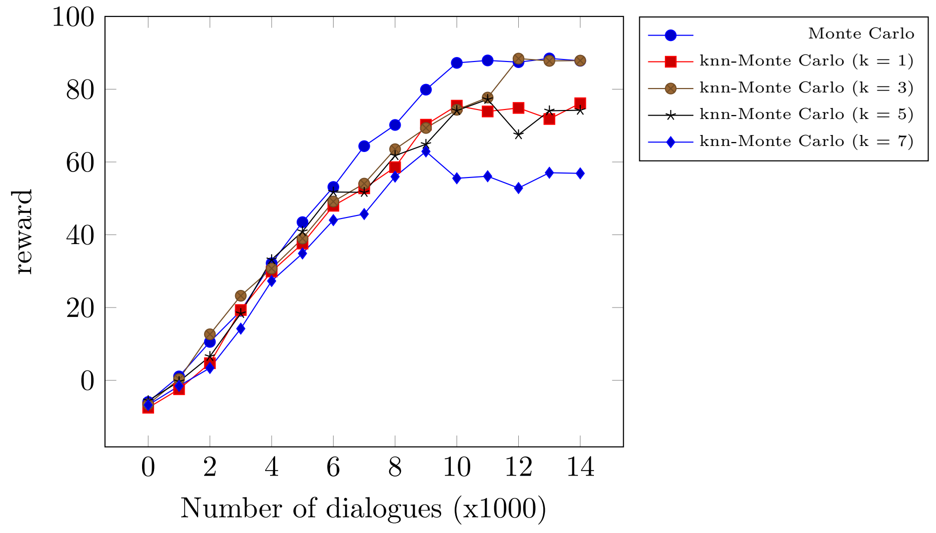


Figure 5.8.: Influence of the training algorithm on the dialogue reward.

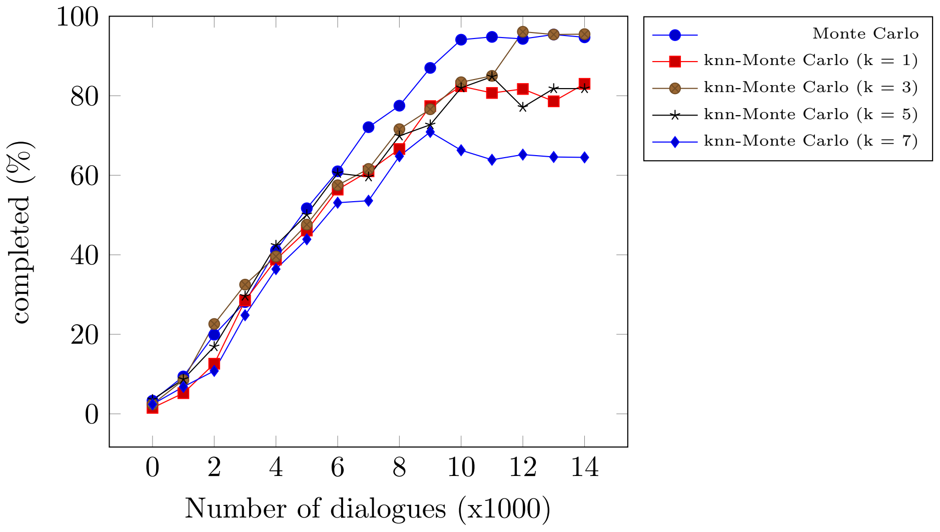


Figure 5.9.: Influence of the training algorithm on the number of successfully completed dialogues.

and is labelled the *Handcrafted* confusion model.

Multiple experiments were conducted with an increasing error rate to determine the system's robustness to different noise levels. Although the resulting confusions from the *Handcrafted* approach are unlikely to match the type of confusions obtained in live situations, using this method has in fact resulted in relatively effective trained dialogue systems ([TY10]). As can be expected, the system's performance gradually deteriorates when increasing the confusion rate (see Figure 5.11), however using the *Phoneme* approach during training shows a slight improvement of the system's online performance, compared to the *Handcrafted* method. On average it shows an improvement of around 2-3% in successfully completed dialogues and interestingly enough, the benefits of the proposed error channel confusion model scale with the confusion error.

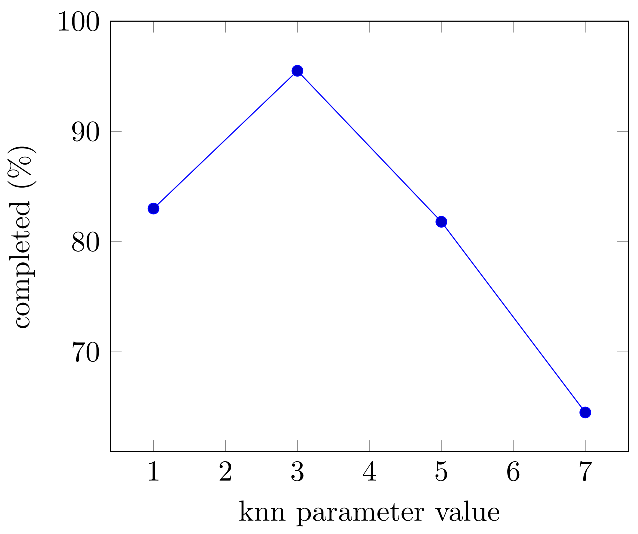


Figure 5.10.: Comparison of different knn values for the Monte Carlo training algorithm.

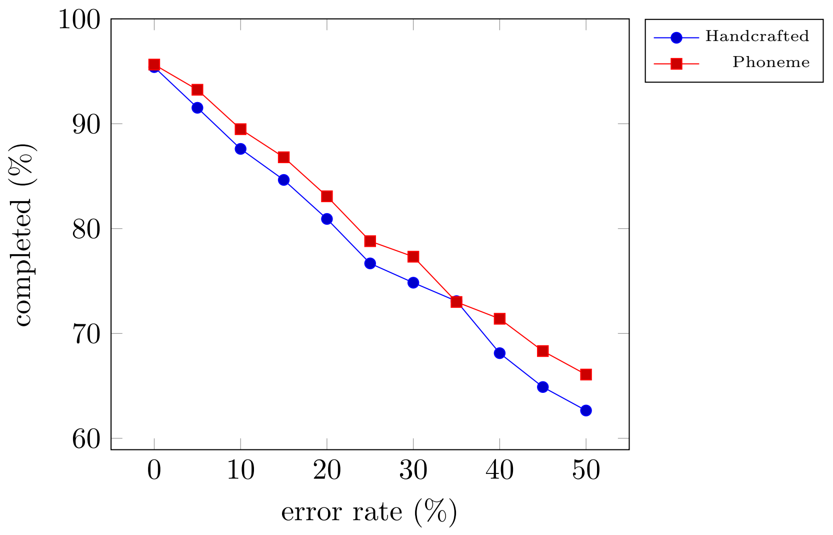


Figure 5.11.: Influence of the confusion model on the number of successfully completed dialogues.

5.3. Summary

From these experiments it can be concluded that the implemented statistical dialogue manager offers a solid foundation for a multitude of spoken dialogue system tasks. The system benefits from being easily configurable and trainable with the only requirement being a valid domain ontology. Furthermore, the non-data driven approach that was employed allows to easily bootstrap dialogue managers (and implicitly spoken dialogue systems) for a multitude of domains with next to no manual work. The success rate of the system in a low-noise environment is around 95% and comparable to similar statistical dialogue managers ([YGK+10]).

An increase in the semantic error rate naturally leads to degradation of the system's performance, however even with 50% error rate the dialogue manager manages to successfully conclude more than 60% of the user tasks. The improvement of noise robustness can be attributed to the error simulation framework used during training, that allows for simulating noisy conditions, thus making the system more flexible in uncertain conditions.

6. Conclusion

6.1. Summary

In this work, a statistical dialogue manager has been developed from scratch with several goals in mind: it should be easily configurable for new SDS tasks, hence no domain information should be hardcoded; it should allow for every component of the dialogue manager to be easily extended or changed by providing a set of interfaces; however, one of the most important goals was to develop a virtual-data driven dialogue manager, thus enabling to bootstrap the system without the need of any real training data.

The dialogue manager uses reinforcement learning (more specifically, the Monte Carlo Control algorithm) to bootstrap the policy that determines the system's responses. Due to the nature of reinforcement learning, several thousand domain specific annotated dialogues are required in order for the system to learn a well performing policy. Manually training the dialogue manager for every scenario will be extremely expensive, even without changes to the core of the framework, which would usually require for the DM to be trained anew. Thus, in order for reinforcement learning to be a viable solution for a statistical dialogue manager, a user simulator component was introduced, which provides automated learning of the policy.

The core of the user simulator is a policy which determines how to interact with the dialogue manager. The policy has been carefully crafted and is driven by several statistical models in order to emulate different types of people interacting with the system.

In order to make the dialogue manager more robust to noise, an error channel has been developed which confuses the input to the dialogue manager during the training process. The error channel uses a confusion model to determine how an incoming semantic representation of a (simulated) user utterance should be distorted. Several different models have already been developed, but they all share the same disadvantage: they require annotated dialogue corpora in order to learn the given confusion model.

In this work, a realistic confusion model has been proposed which does not require any training data. It uses natural language generation to materialize the user utterance, and the sentence in turn is converted into a phoneme graph using a pronunciation dictionary and a phoneme similarity mapping. The resulting graph is then processed by a language understanding component that produces a list of confusions. Various techniques were used to automatically bootstrap the language understanding and language generation components used for the confusion model. The model is also compliant with the overall goal of this work and is scenario agnostic.

The evaluation of the dialogue manager showed a success rate of 95% without any noise and an average of 7 turns for finding a restaurant in a tourist information scenario. The performance of the system is very similar to the HIS model presented in [YGK$^+$10]. Increasing the noise in the environment inevitably decreases the success rate of the system, but even with an error rate of 50% the system manages to complete more than 60% of the dialogues successfully. Additionally, the confusion model proposed in this work, boosts the system's success rate by further 2% compared to the handcrafted model.

6.2. Outlook

While the dialogue manager presented here has been proven to be well performing even without training data, the online performance of the system is also highly dependable on the other components in the SDS pipeline, especially the speech recognizer and the language understanding component. Choosing a suboptimal implementation can greatly reduce the success rate of the system. The language understanding component developed for the phoneme based confusion model can also be used within the SDS pipeline, due to the more conservative setup of the system (keyword, opposing to a CFG parser). However, the language generation component, while sufficient for the confusion model due to the nature of the setup, might not be mature enough for interacting with real users.

Although the dialogue manager can be bootstrapped without any training data, there are certain scenarios that can benefit from training on the system with domain specific annotated dialogue corpora. The dialogue manager implemented as part of this work provides several interfaces for interacting with the training process and thus provides for easy implementation and integration of new training algorithms, including data driven ones. The dialogue state representation can be additionally extended to include more metadata about the system state, which could influence the training process and the performance of the system.

Training data can also be benefitial for the user simulator. While highly configurable, most of the simulator properties are currently manually set by a domain expert, depending on the dialogue manager requirements (e.g. noise levels). Parameters such as user initiative and user patience can be learned from annotated dialogue training data, which can also be from a different domain. Given sufficient training data, the handcrafted user simulator policy can be replaced completely by a learned one. The dialogue manager machine learning algorithms can be reused for learning a user simulator policy.

While the error channel model proposed in this work performs consistently and does not require any training data the performance for large sentences may be very slow. The cause for the performance degradation is the graph size and the subsequent traversal of all nodes. One way to reduce the computational overhead would be to reduce the graph size by not taking into account all neighbour phonemes (see Section 3.2), but only a given subset. An obvious disadvantage of this approach is that it will reduce the number of confusions generated by the model, but performance increase would allow for online usage during training.

Bibliography

[AWD97] M. Araki, T. Watanabe, and S. Doshita, "Evaluating dialogue strategies for recovering from misunderstandings," in *In Proc. IJCAI Workshop on Collaboration Cooperation and Conflict in Dialogue Systems.* Citeseer, 1997, pp. 13–18.

[B$^+$06] C. M. Bishop *et al.*, *Pattern recognition and machine learning.* springer New York, 2006, vol. 1.

[Bli02] L. Blin, "Apprentissage de structures d'arbres à partir d'exemples: application à la prosodie pour la synthèse de la parole," *Université de Rennes*, vol. 1, 2002.

[Bon02] B. Bonet, "An e-optimal grid-based algorithm for partially observable markov decision processes," in *Proc. 19th International Conf. on Machine Learning*, 2002, pp. 51–58.

[BPNZ06] T. H. Bui, M. Poel, A. Nijholt, and J. Zwiers, "A tractable ddn-pomdp approach to affective dialogue modeling for general probabilistic frame-based dialogue systems," 2006.

[Bra97] R. I. Brafman, "A heuristic variable grid solution method for pomdps," in *AAAI/IAAI.* Citeseer, 1997, pp. 727–733.

[BS98] A. G. Barto and R. Sutton, "Reinforcement learning: An introduction. adaptive computation and machine learning," 1998.

[DMA03] Y. Deng, M. Mahajan, and A. Acero, "Estimating speech recognition error rate without acoustic test data." in *INTERSPEECH*, 2003.

[FLAK02] E. Fosler-Lussier, I. Amdal, and H.-K. J. Kuo, "On the road to improved lexical confusability metrics," in *ISCA Tutorial and Research Workshop (ITRW) on Pronunciation Modeling and Lexicon Adaptation for Spoken Language Technology*, 2002.

[GDK94] E. Goldberg, N. Driedger, and R. I. Kittredge, "Using natural-language processing to produce weather forecasts," *IEEE Expert*, vol. 9, no. 2, pp. 45–53, 1994.

[GHL05] K. Georgila, J. Henderson, and O. Lemon, "Learning user simulations for information state update dialogue systems." in *INTERSPEECH*, 2005, pp. 893–896.

[GKM$^+$08] M. Gašić, S. Keizer, F. Mairesse, J. Schatzmann, B. Thomson, K. Yu, and S. Young, "Training and evaluation of the his pomdp dialogue system in noise," in *Proceedings of the 9th SIGdial Workshop on Discourse and Dialogue.* Association for Computational Linguistics, 2008, pp. 112–119.

[GR09] A. Gatt and E. Reiter, "Simplenlg: A realisation engine for practical applications," in *Proceedings of the 12th European Workshop on Natural Language Generation.* Association for Computational Linguistics, 2009, pp. 90–93.

[GY11] M. Gašić and S. Young, “Effective handling of dialogue state in the hidden information state pomdp-based dialogue manager,” *ACM Transactions on Speech and Language Processing (TSLP)*, vol. 7, no. 3, p. 4, 2011.

[HB95] K. Hone and C. Baber, “Using a simulation method to predict the transaction time effects of applying alternative levels of constraint to user utterances within speech interactive dialogues,” in *Spoken Dialogue Systems-Theories and Applications*, 1995.

[HL08] J. Henderson and O. Lemon, “Mixture model pomdps for efficient handling of uncertainty in dialogue management,” in *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers.* Association for Computational Linguistics, 2008, pp. 73–76.

[HP99] E. Horvitz and T. Paek, *A computational architecture for conversation.* Springer, 1999.

[JLK$^+$09] S. Jung, C. Lee, K. Kim, M. Jeong, and G. G. Lee, “Data-driven user simulation for automated evaluation of spoken dialog systems,” *Computer Speech & Language*, vol. 23, no. 4, pp. 479–509, 2009.

[KLC98] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, “Planning and acting in partially observable stochastic domains,” *Artificial intelligence*, vol. 101, no. 1, pp. 99–134, 1998.

[LGHS06] O. Lemon, K. Georgila, J. Henderson, and M. Stuttle, “An isu dialogue system exhibiting reinforcement learning of dialogue policies: generic slot-filling in the talk in-car system,” in *Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Posters & Demonstrations.* Association for Computational Linguistics, 2006, pp. 119–122.

[LGJ$^+$09] F. Lefévre, M. Gašić, F. Jurčíček, S. Keizer, F. Mairesse, B. Thomson, K. Yu, and S. Young, “k-nearest neighbor monte-carlo control algorithm for pomdp-based dialogue systems,” in *Proceedings of the SIGDIAL 2009 Conference: The 10th Annual Meeting of the Special Interest Group on Discourse and Dialogue.* Association for Computational Linguistics, 2009, pp. 272–275.

[LL07] O. Lemon and X. Liu, “Dialogue policy learning for combinations of noise and user simulation: transfer results,” in *Proc. SIGdial*, 2007.

[LPE00] E. Levin, R. Pieraccini, and W. Eckert, “A stochastic model of human-machine interaction for learning dialog strategies,” *Speech and Audio Processing, IEEE Transactions on*, vol. 8, no. 1, pp. 11–23, 2000.

[MWP03] H. M. Meng, C. Wai, and R. Pieraccini, “The use of belief networks for mixed-initiative dialog modeling,” *Speech and Audio Processing, IEEE Transactions on*, vol. 11, no. 6, pp. 757–773, 2003.

[P$^+$96] S. G. Pulman *et al.*, “Conversational games, belief revision and bayesian networks,” in *Computational Linguistics in the Netherlands.* Citeseer, 1996.

[PD06] O. Pietquin and T. Dutoit, “A probabilistic framework for dialog simulation and optimal strategy learning,” *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 14, no. 2, pp. 589–599, 2006.

[PH05] R. Pieraccini and J. Huerta, “Where do we go from here? research and commercial spoken dialog systems,” in *6th SIGdial Workshop on Discourse and Dialogue*, 2005.

[Pie04] O. Pietquin, *A framework for unsupervised learning of dialogue strategies.* Presses univ. de Louvain, 2004.

[Pie06] ——, “Consistent goal-directed user model for realisitc man-machine task-oriented spoken dialogue simulation,” in *Multimedia and Expo, 2006 IEEE International Conference on.* IEEE, 2006, pp. 425–428.

[PR02] O. Pietquin and S. Renals, “Asr system modeling for automatic evaluation and optimization of dialogue systems,” in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, vol. 1. IEEE, 2002, pp. I–45.

[Rei95] E. Reiter, “Nlg vs. templates,” *arXiv preprint cmp-lg/9504013*, 1995.

[RL06] V. Rieser and O. Lemon, “Cluster-based user simulations for learning dialogue strategies.” in *INTERSPEECH*, 2006.

[RML95] E. Reiter, C. Mellish, and J. Levine, “Automatic generation of technical documentation,” *Applied Artificial Intelligence an International Journal*, vol. 9, no. 3, pp. 259–287, 1995.

[RPT00] N. Roy, J. Pineau, and S. Thrun, “Spoken dialogue management using probabilistic reasoning,” in *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics.* Association for Computational Linguistics, 2000, pp. 93–100.

[SB98] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning).* A Bradford Book, Mar. 1998. [Online]. Available: http://www.worldcat.org/isbn/0262193981

[SS73] E. A. Schegloff and H. Sacks, “Opening up closings,” *Semiotica*, vol. 8, no. 4, pp. 289–327, 1973.

[STW$^+$07] J. Schatzmann, B. Thomson, K. Weilhammer, H. Ye, and S. Young, “Agenda-based user simulation for bootstrapping a pomdp dialogue system,” in *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers.* Association for Computational Linguistics, 2007, pp. 149–152.

[STY07a] J. Schatzmann, B. Thomson, and S. Young, “Error simulation for training statistical dialogue systems,” in *Automatic Speech Recognition & Understanding, 2007. ASRU. IEEE Workshop on.* IEEE, 2007, pp. 526–531.

[STY07b] ——, “Statistical user simulation with a hidden agenda,” *Proc SIGDial, Antwerp*, pp. 273–282, 2007.

[SWSY06] J. Schatzmann, K. Weilhammer, M. Stuttle, and S. Young, “A survey of statistical user simulation techniques for reinforcement-learning of dialogue management strategies,” *The Knowledge Engineering Review*, vol. 21, no. 02, pp. 97–126, 2006.

[SWY04] M. N. Stuttle, J. D. Williams, and S. Young, “A framework for dialogue data collection with a simulated asr channel.” in *INTERSPEECH*, 2004.

[SY02] K. Scheffler and S. Young, “Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning,” in *Proceedings of the second international conference on Human Language Technology Research.* Morgan Kaufmann Publishers Inc., 2002, pp. 12–19.

[TGK+08] B. Thomson, M. Gasic, S. Keizer, F. Mairesse, J. Schatzmann, K. Yu, and S. Young, “User study of the bayesian update of dialogue state approach to dialogue management.” in *INTERSPEECH*, 2008, pp. 483–486.

[Tra99] D. R. Traum, “Computational models of grounding in collaborative systems,” in *Psychological Models of Communication in Collaborative Systems-Papers from the AAAI Fall Symposium*, 1999, pp. 124–131.

[TSW+07] B. Thomson, J. Schatzmann, K. Weilhammer, H. Ye, and S. Young, “Training a real-world pomdp-based dialogue system,” in *Proceedings of the Workshop on Bridging the Gap: Academic and Industrial Research in Dialog Technologies.* Association for Computational Linguistics, 2007, pp. 9–16.

[TSY08] B. Thomson, J. Schatzmann, and S. Young, “Bayesian update of dialogue state for robust dialogue systems,” in *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on.* IEEE, 2008, pp. 4937–4940.

[TY10] B. Thomson and S. Young, “Bayesian update of dialogue state: A pomdp framework for spoken dialogue systems,” *Computer Speech & Language*, vol. 24, no. 4, pp. 562–588, 2010.

[TYK+10] B. Thomson, K. Yu, S. Keizer, M. Gasic, F. Jurcícek, F. Mairesse, and S. Young, “Bayesian dialogue system for the let’s go spoken dialogue challenge,” in *Spoken Language Technology Workshop (SLT), 2010 IEEE.* IEEE, 2010, pp. 460–465.

[VR08] C. Venour and E. Reiter, “Tutorial for simplenlg (version 3.7),” 2008.

[WAD98] T. Watanabe, M. Araki, and S. Doshita, “Evaluating dialogue strategies under communication errors using computer-to-computer simulation,” *IEICE transactions on information and systems*, vol. 81, no. 9, pp. 1025–1033, 1998.

[Wal00] M. A. Walker, “An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email,” *Journal of Artificial Intelligence Research*, vol. 12, pp. 387–416, 2000.

[Wil10] J. D. Williams, “Incremental partition recombination for efficient tracking of multiple dialog states,” in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on.* IEEE, 2010, pp. 5382–5385.

[WRR02] M. A. Walker, O. C. Rambow, and M. Rogati, “Training a sentence planner for spoken dialogue using boosting,” *Computer Speech & Language*, vol. 16, no. 3, pp. 409–433, 2002.

[WY07a] J. D. Williams and S. Young, “Partially observable markov decision processes for spoken dialog systems,” *Computer Speech & Language*, vol. 21, no. 2, pp. 393–422, 2007.

[WY07b] ——, “Scaling pomdps for spoken dialog management,” *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, no. 7, pp. 2116–2129, 2007.

[YGK+10] S. Young, M. Gašić, S. Keizer, F. Mairesse, J. Schatzmann, B. Thomson, and K. Yu, “The hidden information state model: A practical framework for pomdp-based spoken dialogue management,” *Computer Speech & Language*, vol. 24, no. 2, pp. 150–174, 2010.

[You07] S. Young, “Cued standard dialogue acts,” *Report, Cambridge University Engineering Department, 14th October*, vol. 2007, 2007.

[YSWY07] S. Young, J. Schatzmann, K. Weilhammer, and H. Ye, “The hidden information state approach to dialog management,” in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 4. IEEE, 2007, pp. IV–149.

Appendix

A. Domain Ontology

```
entity -> venue(type, name, area, near, addr, phone, postcode)
type -> placetostay(staytype, hasinternet, hasparking, price, pricerange, stars)
type -> placetoeat(eattype, pricerange, openhours, price)
type -> placetodrink(drinktype, pricerange, openhours, price)
type -> placetosee(seetype, pricerange, openhours)
type -> entsvenue(entstype)
type -> univenue(unitype, openhours)
type -> sportsvenue(sport)
type -> transvenue(transtype)
type -> shopvenue(shoptype, openhours)
type -> amenity(amtype)
amtype -> hospital()
amtype -> policestation()
amtype -> bank(openhours)
amtype -> postoffice(openhours)
amtype -> touristinfo(openhours)
shoptype -> supermarket()
shoptype -> shoppingcentre()
transtype -> airport()
transtype -> busstation()
transtype -> trainstation(openhours)
staytype -> guesthouse()
staytype -> hotel()
eattype -> restaurant(food)
drinktype -> bar(childrenallowed, hasinternet, hasmusic, hastv, openhours, price)
drinktype -> coffeeshop()
drinktype -> pub(childrenallowed, hasfood, hasinternet, hastv)
seetype -> architecture()
seetype -> museum()
seetype -> park()
unitype -> college()
unitype -> library()
entstype -> cinema()
entstype -> theatre()
food = { American, Cafe food, Chinese }
pricerange = { free, cheap, moderate }
sport = { badmintoncourt, cricketfield, footballfield }
```

Figure A.1.: Domain ontology

B. Sample user simulator dialogue

$$C_0 = \begin{bmatrix} type = bar \\ drinks = beer \\ area = central \end{bmatrix}$$

$$R_0 = \begin{bmatrix} name = \\ addr = \\ phone = \end{bmatrix}$$

Sys 0 Hello, how may I help you?

$$A_1 = \begin{bmatrix} inform(type = bar) \\ inform(drinks = beer) \\ inform(area = central) \\ request(name) \\ request(addr) \\ request(phone) \\ bye() \end{bmatrix}$$

Usr 1 I am looking for a fine beer bar.

Sys 1 Ok, a wine bar. What pricerange?

$$A_2 = \begin{bmatrix} negate(drinks = beer) \\ inform(price = cheap) \\ inform(area = central) \\ request(name) \\ request(addr) \\ request(phone) \\ bye() \end{bmatrix}$$

Usr 2 No, beer please!

Sys 2 You are looking for a beer bar, correct?

$$A_3 = \begin{bmatrix} affirm() \\ inform(price = cheap) \\ inform(area = central) \\ request(name) \\ request(addr) \\ request(phone) \\ bye() \end{bmatrix}$$

Usr 3 Yeah something cheap in the town centre.

Sys 3 Murphy's on Main Square serves cheap beer.

$$A_4 = \begin{bmatrix} request(phone) \\ bye() \end{bmatrix}$$

Usr 4 Ok, and what' s the phone number?

Sys 4 The number is 796 69 94.

$$A_5 = \begin{bmatrix} bye() \end{bmatrix}$$

Usr 5 Thanks, goodbye!

Figure B.2.: Sample dialogue and agenda sequence

C. ARPAbet phonemes

Phoneme	Example	Translation
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
IH	it	IH T
IY	eat	IY T
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
OW	oat	OW T
OY	toy	T OY
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
UH	hood	HH UH D
UW	two	T UW
V	vee	V IY
W	we	W IY
Y	yield	Y IY L D
Z	zee	Z IY
ZH	seizure	S IY ZH ER

Table C.1.: ARPAbet phonemes

D. ARPAbet phoneme neighbours

Phoneme	Neighbours
IY	IY|IH|IX
IH	IH|IY|AX|EH
EH	EH|IH|AX|ER|AE
AE	AE|EH|ER|AH
AH	AH|AE|ER|AA
AA	AA|AH|ER|AO
AO	AO|AA|ER|AX|UH
UH	UH|AO|AX|UW|UW
UW	UW|UH|AX|UW
UW	UW|IX|AX|UH|UW
IX	IX|IY|IH|AX|UW
AX	AX|IX|ER|UW
ER	ER|EH|AH|AO|AX
EY	EY|EH|IY|AY
OY	OY|AO|IY|AY
AY	AY|AA|IY|OY|EY
AW	AW|AA|UH|OW
OW	OW|AO|UH|AW
P	P|T|B|HH
T	T|CH|K|D|P|HH
CH	CH|SH|JH|T
K	K|G|T|HH
SH	SH|S|ZH|CH
S	S|SH|Z|TH
B	B|P|D
D	D|T|JH|G|B
JH	JH|CH|ZH|D
G	G|K|D
ZH	ZH|SH|Z|JH
Z	Z|S|DH|ZH
TH	TH|S|DH|F|HH
F	F|HH|TH|V
DH	DH|TH|Z|V
V	V|F|DH
HH	HH|TH|F|P|T|K
L	L|R|W
R	R|Y|L
Y	Y|W|R
W	W|L|Y
M	M|N
N	N|M|NG
NG	NG|N
DX	DX|DX|S|HH|DH
DX	DX|DX|T
N	N|AX|M|NG

Table D.2.: ARPAbet phoneme neighbours map