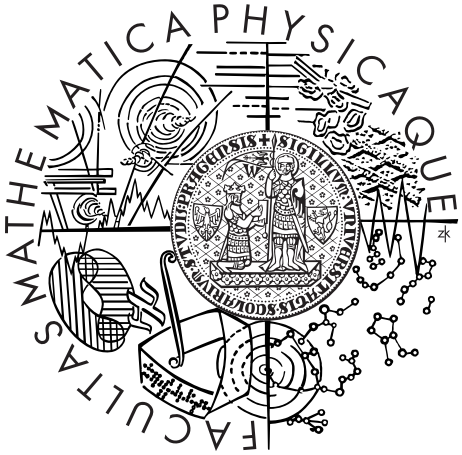
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Vejman

Development of an English public transport information dialogue system

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: Mgr. Ing. Filip Jurčíček Ph.D.
Study programme: Informatics
Specialization: Theoretical Computer Science

Prague 2015

I thank my supervisor Mgr. Ing. Filip Jurčíček, Ph.D. for his patience and kind critical comments that directed my effort while working on this thesis.
I also thank my mother for her continuous support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Development of an English public transport information dialogue system

Autor: Martin Vejman

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí diplomové práce: Mgr. Ing. Filip Jurčíček, Ph.D.

Abstrakt:

Tato práce se zabývá vývojem anglického dialogového systému, který je založen na frameworku Alex určeném pro vytváření dialogových systémů. Práce popisuje adaptaci komponent frameworku na novou doménu a anglický jazyk. Výsledný dialogový systém poskytuje informace o veřejné dopravě ve městě New York. Součástí práce je příprava statistického modelu a nasazení vlastního rozpoznávače řeči pomocí nástrojů Kaldi. Bylo s ním ve srovnání s Google Speech API dosaženo lepších výsledků, které vychází ze subjektivního hodnocení uživatelů získaného pomocí crowdsourcingu.

Klíčová slova: strojové učení, automatické rozpoznávání řeči, dialogový systém

Title: Development of an English public transport information dialogue system

Author: Martin Vejman

Department: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Ing. Filip Jurčíček, Ph.D.

Abstract: This thesis presents a development of an English spoken dialogue system based on the Alex dialogue system framework. The work describes a component adaptation of the framework for a different domain and language. The system provides public transport information in New York. This work involves creating a statistical model and the deployment of custom Kaldi speech recognizer. Its performance was better in comparison with the Google Speech API. The comparison was based on a subjective user satisfaction acquired by crowdsourcing.

Keywords: machine learning, automatic speech recognition, spoken dialogue systems

Contents

Introduction

Providing information to consumers is a common task, usually solved by implementing a web page or a mobile application. An alternative to these approaches is a spoken dialogue system. It allows people to interact with a computer in the most natural way, by voice. Spoken dialogue system is intuitive, direct and hands-free, which renders an opportunity for deployment in many fields. There is no need to find a mobile device and enter the queries into a puzzling interface, it is enough to simply say it. Because of this, it may serve as a valuable asset for obtaining information for the elderly and especially for blind people and people with visual impairment [1].

However, it is very difficult to implement a dialogue system from ground up. Fortunately the Alex Spoken Dialogue Framework (ASDF) incorporates all of the key components needed for performing the task [2]. We decided to take the advantage of ASDF and implement a spoken dialogue system in English for providing Public Transport Information in New York (PTINY). New York, is a city with the highest ridership in the United States and with one of the most extensive subway systems in the world. English presents a wider range of use and larger competition to be compared with. This enables us to build better systems with greater potential.

Automatic Speech Recognition (ASR) is not yet at the point where every utterance could be reliably identified by a computer [3]. However, it is possible to achieve substantially better results by using a speech recognizer within a limited domain [4]. A crowdsourcing platform CrowdFlower is used to evaluate our Public Transport Information (PTI) solution and to collect speech data. The collected speech data are used to train a Kaldi speech recognizer and it is compared with the Google ASR. The comparison measures are based on the subjective user satisfaction of CrowdFlower contributors.

Our PTI solution is a useful showcase of the ASDF and it will further contribute to collecting speech data for improving the quality of speech recognizers. It also participated in the Metropolitan Transportation Authority (MTA) App Quest competition.

The main results of this thesis are the public transport information phone-based dialogue system for New York (PTINY) and a Kaldi decoder for English speech recognition.

This thesis is organized as follows. In the first chapter we look at the used technologies. In the second chapter we discuss the implementation in its principles by each component. The third chapter covers the features of the implemented solution and displays its capabilities. The fourth chapter describes the workflows associated with creating a dialogue system along with training language model and building a Kaldi decoder. In the fifth chapter we compare the Kaldi ASR with the cloud-base ASR by Google and summarize the findings.

1. Technologies used

In this chapter we introduce main technologies that were involved in creating our dialogue system.

1.1 Alex spoken dialogue framework

Alex Spoken Dialogue Framework (ASDF)[1] serves for utilizing research in the development of spoken dialogue systems. It is maintained by the dialogue systems group at Institute of Formal and Applied Linguistics (UFAL) [2], Faculty of Mathematics and Physics, Charles University in Prague. And it is written in Python.

The ASDF consists of baseline components for assembling spoken dialogue systems. There are tools for processing logs and evaluating spoken dialogue systems. These tools can be used for audio transcriptions or semantic annotation for example. A small set of example dialogue system implementations for different domains is also present.

There is a working Public Transport Information (PTI) [12] for Prague public transport and the Czech Republic transport network in Czech language. Our solution is based on the Czech version. However, switching to English renders many challenges emerging from culture or speech habit differences for example. It brings the advantage of having more versatile system deployable in different cities or countries just by changing a knowledge base. Collecting English data is also important for creating better models that can be facilitated by other applications within the ASDF.

Automatic Speech Recognition

Automatic Speech Recognition (ASR) transforms speech into text. Many applications already use ASR technology as an interface between human and a computer, although it is not yet capable of understanding all speech in any environment. Many factors influence perception of voice.

Acoustic conditions, voice differences, distance from the recording device, heavy accent, even voice emphasis, these are few of the issues versatile ASR has to cope with. Very good overall performance delivers cloud-based speech recognition Application Programming Interface (API) by Google which can be utilized withing the ASDF. Achieving better quality requires many hours of transcribed text.

However, when we restrict the recognition scope to a specific domain, the amount of words the recognizer needs to handle becomes quite limited. There is only so many expressions that can be used in a common conversation about particular subject. With a recognizer trained on narrower domain, better results can be achieved.

Kaldi is an open-source [3] toolkit for speech recognition based on finite state

[1] `https://github.com/UFAL-DSG/alex`

[2] `https://ufal.mff.cuni.cz/grants/vystadial`

[3] Apache License 2.0

transducers. We use python wrapper Pykaldi[4] within the ADSF for building a Kaldi decoder and effectively deploying trained ASR [5]. Kaldi decoder requires statistical models – an Acoustic Model (AM) and a Language Model (LM). The AM is trained within the department [6]. It defines probabilities of acoustic features for a given word. The LM is more domain specific as it refines probabilities of a word being recognized. We use ASDF scripts that utilize the SRI Language Modeling Toolkit (SRILM)[5] for training LMs. The process of training AM and testing Kaldi will be expanded upon in Chapter 4 on page 19.

Voice Activity Detection

We need to be able to determine the end of the utterance for the computer to take turn and respond. This role is performed by the Voice Activity Detection (VAD) component. VAD cuts speech into sentences which are sent to ASR for processing. It separates noise and silence from the speech.

Text To Speech

Text to Speech (TTS) makes an instantaneous impression as this is the first and in most cases the only output end user is able to perceive. The ASDF supports multiple TTS alternatives, Google, Flite, SpeechTech and VoiceRSS. We have utilized VoiceRSS[6], the free online service. The VoiceRSS API requires API key which is limited to per day requests. With the ASDF caching most of the web requests, it suffices our intents.

VoIP interface

Our spoken dialogue system communicates with users over a phone. The ASDF exploits a modified version of communicating library PJSIP[7] for implementing VoIP applications. There is no need for registering a telephone number, for running a dialogue system it is suffice to enter Session Initiation Protocol (SIP) account details. SIP account can be freely registered at numerous providers.

For accepting incoming calls from USA, a toll-free number was provided by the department.

1.2 Crowdsourcing

Crowdsourcing is a method for acquiring data by delegating work to a community of people. In particular online communities tend to be employed for their convenience. By dividing tasks into smaller independent parts, one can eliminate the need for expert workers and therefore reduce costs associated with acquisition of the coveted data. In some cases the cost savings can be a tenfold of what in-house solution may provide as mentioned in [7]. However, this method can barely achieve the quality or accuracy of the expert workers.

[4] `https://github.com/UFAL-DSG/pykaldi`
[5] `http://www.speech.sri.com/projects/srilm/`
[6] `http://www.voicerss.org/`
[7] `https://github.com/UFAL-DSG/pjsip`

Collecting speech data for training ASR models in English is easier with the help of crowdsourcing. There are several crowdsourcing platforms connecting workers with work requesters such as Amazon Mechanical Turk[8], Samasource[9], CrowdFlower[10] and many more.

Samasource is a non-profit organization with a noble cause of lifting people out of poverty through digital work [8]. It does not, however, meet our need of employing native English speakers. While Amazon Mechanical Turk would match our requirements, it is no longer available for non-US requesters. With CrowdFlower, we are able to implement a custom solution directly within the platform.

CrowdFlower has mechanisms such as monitoring answer distributions and computing confidence score for maintaining quality of the output data. They claim great amount of contributor force which promises prompt job resolution. The platform contains comprehensible templates for common tasks. It features a web interface for building a custom job from scratch with a sensible support and demonstrative examples, too.

1.3 Deployment

Running a real-time dialogue system can claim considerable amount of system resources. All of the components of the dialogue system in the ASDF are separate processes. Also the static knowledge-base may outgrow ordinary computer's memory. Hence, it is only right to employ multi-processor machine with sufficient amount of memory.

1.3.1 Docker

Docker[11] is a platform for rapid deployment of applications. It contains a packaging tool and a lightweight runtime. An application wrapped in docker is easily portable to any laptop or Virtual Machine (VM). The chain of events leading from discovering a flaw and changing a source code, to deploying compiled dialogue system, can be excessively accelerated with docker.

Dockerfile is a specification file used for automating docker image builds. It allows to specify a sequence of instructions executed on a base image in order to create a new one. Built docker image with the dialogue system can be executed in an isolated container.

The Alex Spoken Dialogue Framework (ASDF) is now using docker which is very useful for resolving dependencies. Our Dockerfile is based on the base ASDF Dockerfile with instructions for downloading newest models and knowledge-base. There is a `-i` flag for mounting any directory into a docker container.

[8] `https://www.mturk.com/mturk/welcome`
[9] `http://samasource.org/`
[10] `http://www.crowdflower.com/`
[11] `https://www.docker.com/`

1.3.2 MetaCentrum

In order to provide service for more than one caller at a time, we need to have multiple instances of our dialogue system. Computing and storage resources of MetaCentrum[12] can be used freely by all students of academic institutions in the Czech Republic. Various VMs were deployed on MetaCentrum for our dialogue system groundwork.

For each instance of MetaCentrum VM, configuration with 4 processors and 16GB was used. Half of the memory would be sufficient, however, the same VMs were employed for training language models which involves excessive memory usage.

[12] `http://www.metacentrum.cz/cs/`

2. Implementation – Public Transport Information in New York

In this chapter we go through each component that needed to be implemented or modified in PTINY. This excludes automatic speech recognition, text to speech and VoIP interface. The principle are described along with relevant instruments. All of these components are domain specific versions of domain independent components of the ASDF.

First we cover keyword database and matching words against the input, than we describe changing states in our dialogue system. Last we explain how the state is translated to the output which concludes in an outline of utterance processing.

2.1 Spoken Language Understanding

To be able to process, evaluate and respond to user's requests, semantic meaning needs to be extracted from utterances. This is realized via Spoken Language Understanding (SLU), which uses a vast static keyword database for analyzing words and phrases of each utterance. Being able to handle such semantic representation makes it possible to change state of the dialogue system. There are different approaches for SLU development. There are SLU techniques based on statistical models learned from data. We, however, have implemented a handcrafted SLU based on simple keyword rules. Both approaches are supported by the ASDF as demonstrated in [9].

Keyword database

Public transport information domain demands the ability to respond to two major constraint queries - location and time. The time and supplementary keywords can be defined explicitly or they can be generated by a simple script. However, the location data are specific for the region we decided to cover and therefore must be gathered.

We ultimately need just the name of the waypoint for the keyword matching process. However, the stop or city names might be ambiguous which is why we need to keep further knowledge about geographic information and more general area.

Location data terminals

The types of waypoints are streets, stops, boroughs, cities and states. All of these location categories are listed in a separate file for the convenience of adding new or updating existing entries. It is obvious that the borough list will be very narrow and may be unnecessary because there is only five boroughs in New York. But the idea is to be able to distinguish between streets and stops with the same name that are very likely to appear within the same city. If we decided to expand

the system to cover Los Angeles region for example, we might need to add not only LA boroughs but to define a finer administrative division altogether.

In terms of the stops, we collected the latest data from MTA[1], PATH[2], NJ Transit[3], NY Waterway[4] and Amtrak[5] for long-distance trips. Most of the companies are providing their schedules for developers in a unified format. The General Transit Feed Specification (GTFS) defines a common format for public transportation schedules and associated geographic information.[6]

We could adopt the GTFS format which would be very convenient for updating [10]. However, some of the datasets were not strictly following the GTFS making it unfeasible to work with. Missing values, overflowing columns or disunited expressions occurred infrequently, nevertheless throughout notable portion of the data. The benefit of an easy update and access to additional information did not outweigh the shortcomings encountered.

We opted for a simple format that takes only the most important features into account. Selecting fewer columns makes it easier to add places that are not available in the GTFS. This includes a few smaller transport companies commuting between tens of terminals that we also included into our database.

In addition to official stops, we added over a hundred of the most popular sites in New York from various top n lists. Those can be used as good reference points in everyday commutes so that the following sentence can be handled for instance.

"From Empire State Building to Central Park.".

We used Google Geolocation API[7], to obtain longitude, latitude and borough for each popular site. Geo coordinate information is preferably used when looking for a connection.

The obtained data in raw form can not be used for keyword matching. As opposed to the Czech language, there is no necessity to take inflection into account, however, there is a number of ways to express stop or a street. Stops in particular, mostly called after an intersections, can be unfolded and expressed in different order and coupled with a different conjunction.

For example the stop `1 Av/E 111 St` can be expressed as

"east hundred eleventh street first avenue".
"east hundred eleventh street at first avenue".
"east hundred eleventh street and first avenue".
⋮
"east one hundred and eleventh street at first avenue".
⋮
"first avenue and east hundred and eleventh street"
⋮

[1]Metropolitan Transportation Authority - `http://www.mta.info/`
[2]Port Authority Trans Hudson - `http://www.panynj.gov/`
[3]New Jersey Transit - `http://www.njtransit.com/`
[4]New York Waterway - `http://www.nywaterway.com/`
[5]The National Railroad Passenger Corporation - `http://www.amtrak.com/home`
[6]GTFS - `https://developers.google.com/transit/gtfs/`
[7]`https://developers.google.com/maps/documentation/business/geolocation/`

The raw data contain numbers and unpronounceable characters like parenthesis, slashes, dashes and also abbreviations that are not unified. For example the `St` can mean both *street* and *saint* and to continue, the word *expressway* is abbreviated by `ep`, `ex`, `exp`, `expy` and `expwy`. Thus for each category we have a separate file with possible forms generated by an expansion script.

Dialogue Act Scheme

Intents of the user as well as actions of the spoken dialogue system are represented by Dialogue Act (DA). They consist of one or more Dialogue Act Items (DAIs) that are elementary semantic information units.

DAIs are defined by a type, slot name and slot value. The slot name and value are domain specific and further define the meaning. In our case slot names may refer to a place or time for instance. Exemplary DA is shown in table 2.1. We can see that the *When does* and *leave* correspond with the request DAI and that the `inform` is gathered from the word *bus* in the sentence.

Utterance	*"When does the bus leave?"*
Dialogue Act	`request(departure_time)&inform(vehicle="bus")`

Table 2.1: Example of semantic notation of an utterance

Sometimes it is not clear how an utterance should be transformed into DA due to unknown context or ASR lapse. The ASDF contains a dialogue act confusion network that deals with this issue. The confusion network stores a probability for each DAI and it presents the most likely DA based on the probability distribution of DAIs.

A confusion network is best utilized when processing ASR n-best hypothesis and using statistical SLU.

Handcrafted SLU

Handcrafted SLU handles only the 1-best hypothesis from ASR. After an utterance is passed into handcrafted SLU, it is matched against class labeled database keywords and an abstract utterance marked with labels is produced. Each label corresponds with a special parsing procedure that yields dialogue acts into the dialogue act confusion network. The following class labels have their designated routines.

- `NUMBER` - Parsing hour and minute values and time fractions.
- `PLACE` - Parsing waypoints from stop, street, borough, city and state values.
- `TIME` - Absolute and relative time periods matching.
- `TASK` - Conversation topic which is either weather, current time or finding connection.
- `VEHICLE` - Preferred means of transport matching.

Due to the iconic Manhattan street grid, people in New York are likely to know their position based on the street and avenue names which are commonly numbers. They may not know the closest bus or subway station. Therefore we decided to support streets as valid input for finding connections. The idea is to let users specify an intersections rather than stops. Stops however, make for more accurate search queries because latitude and longitude values are associated with them. The ambiguity of streets and stops is not negligible, hence boroughs are also parsed as waypoint entries.

Further series of matching steps take place after those routines. Keywords and phrases are being searched for in the whole utterance regardless of the context. This yields more DAIs to the dialogue act confusion network by a simple if-else set of rules. It handles particular utterances for courtesy, greeting, acknowledgement as well as requests and notifications about public transport restrictions. Also DAIs from non-speech events like silence or noise DAIs are extracted here.

2.2 Dialogue Manager

Dialogue Manager (DM) is a component responsible for processing and changing dialogue states in order to take appropriate actions in response to the user's query. The history of the dialogue and inner states are recorded for better comprehension of current request.

Ontology

The ontology contains a static domain knowledge information that can be used for better understanding relations between entities. It defines DAI slot types and values from keyword database and relationships between them. This allows DM to gain more relevant information for example by context resolution.

In addition, it provides relations between locations for discovering compatibility conflicts and for implicit value inference. The compatibility lists are bidirectional and concern street-borough, stop-borough and city-state relations.

Handcrafted DM

Our implementation of handcrafted DM extracts facts from the combination of inner states, history of the dialogue and the DAI probability distribution taken over from the dialogue act confusion network from SLU. From an if-else rule block, it selects a subroutine for deciding what will be the next action taken.

Simple responses to elementary facts are among the first served by the rule block. Those include actions for greeting, repetition of the last system utterance, a context specific help or reseting the system. In case the input yields no change since the last time, or the input from ASR was invalid, the DM executes a back-off action, which is randomly selected from providing help, repeating the last utterance, silence and dispatching an act for saying it simply did not understand.

2.3 Natural Language Generation

Natural Language Generation (NLG) component transforms inner states of the dialogue system into readable text form. Limited domain relieves the amount of DA necessary to transform, therefore we are able to cover NLG by a template dictionary that has entries for each dialogue act item and combinations of some dialogue act items. Seamless communication can be achieved by constructing adequate NLG templates. The slot value of each DAI is treated as a variable that can be inserted in the translated sentence. An example of NLG translation is displayed at 2.2. It is evident how the slot value *Broadway* is injected into the template and also that the time value is translated to word representation.

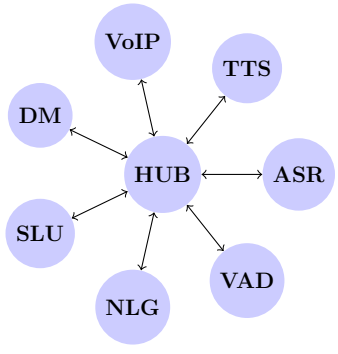
Dialogue Act	`inform(to_stop="Broadway")&inform(arrival_time="04:26:PM")`
NLG template	`inform(to_stop={to_stop})&inform(arrival_time={arrival_time}):` `"It arrives at {to_stop} at {arrival_time}."`
NLG output	*"It arrives at Broadway at four twenty six P M."*

Table 2.2: Translation example of dialogue act to sentence by Natural Language Generation component

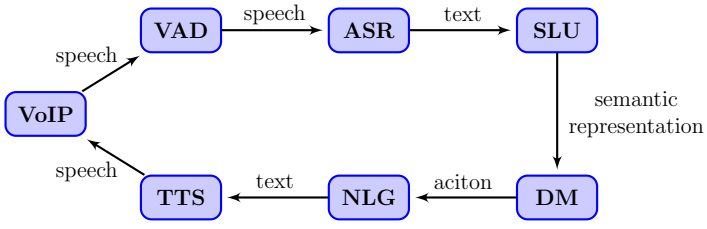
There can be multiple expressions defined for each dialogue act, which is useful for making overused dialogue acts, such as greetings, seem more natural and less robotic. The NLG templates can be overlapping and proper translation rule has to be searched for. The search proceeds from exact to general and from long to short sequences of dialogue act items.

2.4 Main System Hub

The central hub gives the dialogue system modularity. All of the components are connected together via main hub in a star-like shape shown in figure 2.1a. Each component runs as a separate process and the hub essentially chains them via standard stream pipelines as shown in 2.1b and coordinates their continuity.



(a) Hub configuration



(b) Component chain

Figure 2.1: On the left there is a typical star-like shape configuration of the dialogue system components and the right figure shows the inner component chain of the central hub.

3. Features – Public Transport Information in New York

In this chapter we describe the functionality of PTINY as a whole. The features are derived from the DM capabilities which is the brain of the dialogue system, but each component has to oblige. When describing how the dialogue system responds to a particular request, the SLU has to extract semantics from the text input, DM has to decide what to do and NLG has to generate text from the dialogue acts.

3.1 Providing current time

The user can make better route selection decisions based on the knowledge of current time. This is why it is important for the system to be able to provide it. As opposed to the Czech Republic, there are places at different time zones in the United States. Therefore we decided to support current time queries specifying a city or a state for providing more accurate, localized time. The state is enough information to receive a location specific time, however specifying a city is more accurate as some states occupy two timezones. If only a city is specified, the DM will respond with a dialogue act requesting the state name, unless the city is not ambiguous. In which case it will infer the state from the ontology. An example of current time inquiry is shown in table 3.1.

Speaker	**Utterance**
`User`	*"What is the time now in California?"*
`System`	*"In California, it is 7:01 AM, Pacific Daylight Time."*

Table 3.1: Example dialogue for obtaining current time in California.

The time zone data are received from the Google Time Zone API[1]. If the API is inaccessible for some reason it returns an apology act with default computer time set for New York, which is in Eastern Time Zone. Time zone names could be included in the ontology as additional data, however this way, there is no need to keep track of the daylight saving offsets instituted in some sates.

3.2 Weather forecast

It is comforting for the users to be able to obtain weather information at once. Following the example of utilizing weather forecast in the Czech public transport information, we have implemented an English version. OpenWeatherMap API[2] is used for receiving weather data.

Specifying the city is enough to receive a weather information with the state being inferred from the ontology much like when providing current time, described

[1] `https://developers.google.com/maps/documentation/timezone/`
[2] `http://openweathermap.org/api`

at section 3.1. Except if the city is ambiguous, then it is necessary to specify the state as demonstrated in table 3.3. In addition to city and state, the weather inquiry can be defined by time either relative or absolute.

Speaker	Utterance
User	*"What will be the weather like in five hours?"*
System	*"In five hours It will be forty six degrees and clear sky."*

Table 3.2: Example dialogue for obtaining weather information by relative time specification.

The API allows to ask for the weather at a specific time as well as for the weather forecast for the whole day. In the first example 3.2, there is a weather inquiry specified only by time. Default city of New York is used as a reference point.

Speaker	Utterance
User	*"Tell me the tomorrow weather forecast for Denver."*
System	*"Alright, in Denver tomorrow, please specify the state."*
User	*"Colorado."*
System	*"In Colorado, tomorrow it will be light rain, temperatures forty five to sixty one degrees."*

Table 3.3: Example dialogue for obtaining weather forecast for the next day in Denver, Colorado.

The second example 3.3 shows inquiry of the weather forecast for the next day in Denver, Colorado. The temperature units are in degrees Fahrenheit.

3.3 Finding a connection

The prime asset of PTINY is the ability to effectively respond to transport connection requests.

Connection restrictions

The key restrictions are the location from and where to travel, which is either a city, borough, stop or an intersection of two streets. The ambiguity of a waypoint is resolved in similar manner as in the case of time zone or weather queries. The DM tries to infer the waypoint and returns a request for either more general or more limited area specification if an ambiguity is found. User can further specify the criteria listed in table 3.4.

Both departure and arrival times can be specified in absolute or relative form. The nature of DM allows to say these restrictions at once or one by one which makes for better utilization of the dialogue system. If any of the key restrictions are missing, the system demands appropriate additional information.

After the system responds with a route proposition, the user can either further specify his query or ask about the connection attributes listed in table 3.5.

Criteria	Restrictive utterance example
departure time	*"I want to depart in ten minutes."*
arrival time	*"I want to be there at five o'clock."*
preferred vehicle	*"I want to go by bus."*
transfer count	*"I want to transfer two times at most."*

Table 3.4: Connection search criteria paired with example utterances.

Attribute	Inquiry example
origin	*"Where does it leave from?"*
destination	*"What is the destination?"*
arrival time	*"How long till i be there?"*
departure time	*"When does it leave?"*
transfer count	*"How many transfers are there?"*
transfer duration	*"How long do the transfers take?"*
trip duration	*"How long will i travel?"*
trip distance	*"What is the distance of the trip?"*

Table 3.5: Inquiry examples regarding different attributes of a given connection.

The trips in New York can be very long, hence the sequence of instructions is exhaustive. Thus the length of the trip informs not only about the mileage, but also about the number of stops to pass through before each transfer. Whereas for the duration, system responses only with the time value as demonstrated in table 3.6.

Speaker	Utterance
`User`	*"I want to go from Central Park to Broadway, Queens."*
`System`	*"From Central Park to Broadway in Queens, as a first possibility I have found the following connection. Walk to 5th Av and East 84th St. Take the bus M4 at 1:55 PM from 5th Av and East 84th St towards Midtown Penn Station via Broadway via Fifth avenue. Get off at Fifth Av and West 60 - 59th St. Walk to 5th Av and 59th St. Take the subway N at 2:11 PM from 5th Ave and 59th St towards Astoria, Ditmars Blvd. Get off at Broadway."*
`User`	*"Tell me the distance of the trip."*
`System`	*"The trip is 5.2 miles long, You will travel 10 stops from 5th Ave and East 84th St by bus M4, then it is 5 stops by subway N from 5th Av and 59th St."*
`User`	*"Now, tell me the duration of the trip."*
`System`	*"It takes about 26 minutes."*

Table 3.6: Providing a connection from Central Park to Broadway, Queens is portrayed. Responses to distance and duration inquiries are demonstrated.

The Google Directions API

We use the Google Directions API[3] to acquire connection data. For simple from-to queries we use free API accessible through HTTP and we only use API key for more restrictive queries with preferred vehicle and transfer count limitations. This maximizes the utilization of the key before reaching a monthly fee threshold. The transfer count is not directly mapped to an API request, the offered links are rather filtered by the criteria from the API response. When no connection suites the restrictions, an apology utterance is issued.

Choosing the alternative

Users can browse through offered connections back and forth by requesting next, previous or by saying the explicit number of the connection if the current route does not satisfy their needs. By default, there are four alternative connections at disposal, sorted by departure time.

Alternatives in combination with criteria inquiries

As some route descriptions may be quite long, it is handy to take just a peek at other connection attributes without the need to browse through them. An example of peeking is demonstrated in table 3.7. This can be used prior to selection of particular alternative connection.

Speaker	Utterance
`User`	*"How long does the next link take?"*
`System`	*"The second connection takes about 51 minutes."*
`User`	*"How long till the third connection arrives?"*
`System`	*"The third subway reaches Broadway in 1 hour and 17 minutes."*

Table 3.7: Example of obtaining attributes from alternative connections.

3.4 General cases

The following examples apply to general dialogue. Supporting features are presented.

Conflict apology

If, according to the ontology, the specified city is not located at given state, an apology utterance is produced. Similarly a street or stop to borough or city conflicts are checked. An example is shown at table 3.8. This type of behavior occurs in all of the tasks.

This situation can be settled by two different ways. The conflict can be resolved by negation of the wrong value as shown in 3.9. Or the whole dialogue can be restarted as mentioned later in this chapter.

[3] `https://developers.google.com/maps/documentation/directions/`

Speaker	Utterance
User	*"Tell me the weather forecast for Las Vegas in Iowa"*
System	*"I'm sorry, but the city Las Vegas is not located in the state of Iowa."*

Table 3.8: Example dialogue that produces an apology utterance resulting from incompatibility based on static knowledge from the ontology.

Negation

If the system failed to perceive user's intent correctly or the user decided to change his mind, there needs to be a way to reverse the input. User can explicitly negate the wrong value, which causes a relevant slot in the system to be erased. The system either issues an implicit confirmation of the same slot with a different value from history. Or, if there is not any, it asks about the desired value of the slot as shown in table 3.9.

Speaker	Utterance
User	*"I want to go from Forty Sixth Street."*
System	*"Alright, from Forty Sixth Street, Where are You heading?"*
User	*"You know what, no, not from Forty Sixth Street."*
System	*"Where are You leaving from?"*
User	*"Thirty Sixth Street."*
System	*"You want to go from Thirty Sixth Street, where do You want to go to?"*

Table 3.9: An example of changing mind and subsequent correction of origin waypoint by negation.

Context resolution

The dialogue system needs to be able to interpret user's utterance in the context of previously spoken topic. Partially this is achieved by keeping track of previous states, however, there are situations like the one portrayed in table 3.10, where more sophisticated strategy needs to be engaged.

Speaker	Utterance
System	*"Which stop do You want to depart?"*
User	*"Miami."*
System	*"Alright, from Miami, where do You want to go to?"*

Table 3.10: Example of resolving city from the origin request.

In the example, the system asks about the initial stop. The user replies with a city without a preposition. Even though the system was asking about a stop, the DM needs to deduce that the user does not want to go from particular stop, but rather from the city of Miami. These automatic relations are defined in the ontology.

Orthogonal queries

It is essential for the system to be aware of the previous utterances to produce correct responses. However, as the history is being kept, it may lead to misunderstandings. An obvious confusion may occur when asking about orthogonal queries as showed in table 3.11

Speaker	Utterance
`User`	*"What is the weather like in Saint Louis, Missouri?"*
`System`	*"In , It is eighty degrees and clear sky."*
`User`	*"What is the weather like in Alaska?"*
`System`	*"I am sorry, I don't understand, the city Saint Louis is not located in Alaska."*

Table 3.11: Example of location confusion from the previous utterance.

From the example it is evident that the system did not forget the user mentioned the city of Saint Louis, which was considered relevant in the next turn, where it did not pass the compatibility test based on the knowledge from the ontology. Which of course leads to an apology utterance as mentioned earlier and can be resolved either by negation or by resetting the dialogue.

Selection and confirmation of slot values

When the confusion network in DM contains more values of the same slot that have non-zero probabilities, it has to decide what value is intended by the user. The system will either produce a selection or confirmation utterance based on the probability distribution of the values at particular slot. The selection utterance is challenging the user to decide between two values with same probability. Whereas the confirmation is a yes-no question about a slot with a concrete value.

Reset of the dialogue

The dialogue system may get to a point, where its confusion network contains many uniformly distributed values within the same slot and it asks the user for resolving the correct value. Although the user may consider the matter already closed. This may for example arise when user, after receiving a connection link, wants a different one. The system may no longer be able to appropriately respond to user's requests and altering values by negation is not effective. When such situation occurs, the system can be restarted by saying a phrase *"let's start over"* or *"restart"* or similar phrase indicating new entry. Reseting erases all the slots in the dialogue system and it prompts the user to start asking from the beginning.

Supplementary intents

The dialogue system has to be able to process speech habits commonly occurring in every dialogue. In the table 3.12 there are enlisted all of the supplementary intents of the caller that PTINY is able to process.

Intent	*Cause and action taken by PTINY*
`greeting`	Courtesy act and prompt for inquiries.
`farewell`	Indicates parting and an intent to hang up.
`courtesy`	Usually after satisfactory response it concludes a task. The system will encourage user to ask further questions.
`help requested`	Provides context sensitive help by randomly selecting a subtopic and saying how to specify a appropriate query.
`not understood`	Indistinct ASR input results in an apology and a suggestion to repeat the last utterance.
`silence`	Nothing has been said for a while. The system will ask if caller is still in the presence.

Table 3.12: Response actions for supplementary intents

The help context is based on what the user was talking about in previous turns. If for example the conversation was about finding a connection, it may suggest help in the form shown in table 3.13.

Speaker	**Utterance**
`User`	*"Help."*
`System`	*"You can narrow your search by limiting the number of transfers. Just say, I want a direct connection, for example."*

Table 3.13: Example of context sensitive help utterance.

4. Workflows - Development processes

This chapter is concerned with the process of several procedures repeatedly used while developing spoken dialogue system providing public transport information in New York (PTINY). Very similar approaches might be taken for the development of dialogue systems in different domains.

4.1 Creating CrowdFlower Job

Assembling a CrowdFlower job can be realized through one of many templates for ordinary tasks such as various data analysis, entity annotation, categorization, comparison, revision and many more.Custom and more sophisticated tasks can be carried out from scratch. It is desirable that the tasks are as simple as possible to eliminate errors resulting from the lack of knowledge or misinterpretation.

CrowdFlower provides a web interface for work requesters to edit the task by CrowdFlower Markup Language (CML), CSS and custom JavaScript that runs once on page load. There is a possibility to inject a custom HTML code as well. CML and JavaScript are essential for leveraging Crowdflower's quality control. Both mandatory and optional input controls have to be specified with the CML.

4.1.1 Call job

We created a call job for testing operational dialogue system. Its purpose is to encourage solvers to call on a toll-free number and ask questions about the public transport in New York and to evaluate and rate the system.

To ensure the call is carried out thoroughly by the contributor, we employed a simple generator of four digit codes. A code is handed out by the dialogue system after finishing a call. It is spelled number after number three times over. In the same time, the code is registered at a validation server running on a dedicated MetaCentrum VM. Without this code it is not possible to submit a feedback form and finish the job.

This behavior of the CrowdFlower job is enforced by a CML control with a custom JavaScript validator. When contributor inserts a code to the CML control, the validator sends a request with the code to the validation server. The server compares the code with a set of registered codes from the dialogue system. Only after receiving a positive response, the validator passes. It is unnecessary to match callers identity, this is sufficient measure for enforcing the call.

To further maximize the efficiency, we imposed a rule for the code giveaway. The dialogue system only hands out the validation code after minimum number of turns has passed. This prevents the callers from saying *"Hello, Good bye!"* and collecting the validation code and therefore the reward without fulfilling the task.

The job web page was built as a survey job from scratch. In the premise of the job, we declare four paragraphs concerning the job.

- **Intro** - Introduction to the whole process, mentioning restrictions and remarks.
- **Instructions** - Exact procedure description, how to behave, how to end the call, how to fill the feedback form.
- **Example call** - Demonstrative dialogue between caller and our dialogue system.
- **Consent** - Legal statement concerning the data management and recording the call.

Stops between which caller wants to find a connection are quoted after the premise. Additional question about the link are urged for exploiting the dialogue system features.

A feedback form of subjective user satisfaction concludes the job page. In addition to the mandatory questions an optional field for general comments and field for the validation code are within the form.

A toll-free number provided by the department was used for this job. CrowdFlower allows to geographically limit work force only to United States. The number of calls per job was temporarily set to one to ensure the diversity of callers. Four VMs on MetaCentrum were dedicated to this job to serve multiple callers.

4.1.2 Transcription job

After collecting enough calls, a transcription job was built from a template for audio transcriptions. For each audio track there is a radio button for marking comprehensible tracks and a field for writing transcribed text. Only instructions and data are needed for launching a transcription job.

This kind of job is very common and popular and therefore it is solved by contributors very quickly. However, the contributors differ on spelling of some words and it is absolutely crucial for the job instructions to make it perfectly clear how should the contributor write.

Data are uploaded to CrowdFlower via Comma Separated Values (CSV) file that contains a list of URLs with audio tracks. The default setup suggests to let each track transcribe three times for accuracy. Even more transcriptions yield from setting up dynamic judgments. However, repeated labeling is costly and may tend to move towards the in-house solution in that regard. We decided to keep multiple transcriptions while reducing costs per transcription. The ultimate transcription is decided upon later from the job results by a custom semi-automatic Python script.

CrowdFlower uses test questions for separating the good transcribers from the bad. Test questions in this job are essentially manual transcriptions. We utilized a quiz mode that estimates the quality of a contributor beforehand. It is assembled from test questions and lets only trusted contributors to participate in the job.

In the instructions we defined examples of how common words should be handled. Also a table with symbols for incomprehensible tracks was specified. It is a good practice to let the users know the context. The contributors were

more content when a list of phrases they might hear was included. In our case the list included phrases like *number of transfers*, *duration of the trip*, *weather forecast*, origin and destination stops etc. Even though some of those phrases did not appear in the exact form in the audio tracks, the evidence of improvement was observable in contributor satisfaction stats of the job within CrowdFlower.

4.2 Iterative improvement

At the beginning we had just a vague idea about how the system should behave. We had a general insight of the features from the Czech dialogue system, however we did not know what is the native way of asking for information. Therefore we made a bootstrap list of sentences with their semantic complements, all of which our dialogue system must work on.

When an operational dialogue system was achieved, we employed CrowdFlower workforce for obtaining feedback from real users. Analyzing logs was very important for discovering ways of inquiring information which we did not initially think of. The log analysis and feedback form from CrowdFlower jobs also provided an input on what features are missing or need improvement. The iterative process of improvement is captured in essence by the following steps.

1. Launch a CrowdFlower call job
2. Obtain logs from VMs
3. Fix flaws in:
    - **SLU** - enrich bootstrap from user turns and maintain 100% precision
    - **DM** - amend features of the dialogue system
    - **NLG** - add templates from system turns to polish rough expressions
4. Upload source code to VMs
5. Restart the dialogue systems.

The dialogue system on each VM is running in a docker container. Any folder can be mounted to the docker container via `-v` flag. Uploading source code to update dialogue system on VM is therefore effortless and makes the development loop very quick.

4.3 Building Kaldi ASR

For building Kaldi decoder we used Pykaldi[1] docker image containing the essential tools. It is necessary to add dependencies for ASDF if building and evaluation is intended within the platform. SRILM[2] must be installed for training Language Model (LM).

[1] `https://github.com/UFAL-DSG/pykaldi`
[2] `http://www.speech.sri.com/projects/srilm/`

Prior to training LM, it is necessary to dump database for creating a labeled list of database entries. It is used for balancing probabilities of every database entry within its class in the LM. Finally, we need to define domain specific corpus for training the LM . Our training data consist of utterances from CrowdFlower call logs, bootstrap utterances and utterances generated by grammar.

Context-free grammar

Creating a good LM entails a good probability distribution of words in the corpus. This can be achieved naturally by collecting a lot of transcriptions. As we do not posses large number of transcriptions, we decided to bootstrap LM with generating utterances by grammar. It should produce utterances that are most likely to be used and therefore it should cover the most frequent cases.

Our context-free grammar is written in Python and it can be assembled from the following prescriptions for simple rewriting rules.

- **Alternative** – exactly one of many

  $A^i(x_1, x_2, ..., x_n)$ adds $A^i \to x_1$
  $A^i \to x_2$
  $\vdots$
  $A^i \to x_n$

- **Option** – either present or not

  $O^i(x)$ adds $O^i \to x$
  $O^i \to \lambda$

- **Sequence** – chain of rules

  $S^i(x_1, x_2, \ldots, x_n)$ adds $S^i \to x_1\, x_2\, \ldots\, x_n$

where for the i-th rewriting rule:

$$\begin{aligned} &\left\{A^i, O^i, S^i\right\} \subseteq V_N \ldots \text{nonterminals} \\ &x, x_1, x_2, \ldots x_n, \lambda \in V_T \ldots \text{terminals} \\ &n \in \mathbb{N} \end{aligned}$$

Explicit grammar can be assembled using these prescriptions which can than simply generate random utterances in desired number. An example of plain grammar can be built as follows.

```
pref_p = A('can you tell me', 'i would like to know')
pref_q = A('what is', 'what will be')
subj = A('weather', 'forecast', 'weather forecast')
period = A('tomorrow', 'in the afternoon')
weather = S(O(pref_p), O(pref_q), 'the', subj, O(period))
```

The nonterminal `weather` yields utterances asking about the weather. Terminals can be also loaded from file, which is useful for defining alternatives for waypoints for example.

The final grammar should cover as many utterances as possible. However, it is easy to include utterances that are not used in conversation or does not make sense at all. From the example above, the utterance *"can you tell me the forecast tomorrow"* is not exactly what we wanted to include. Even though it is syntactically correct, it is not something to be used in PTI domain. This is undesirable to have in our corpus.

In addition to these rules, we have added a possibility to add explicit probability with which the rule should be selected. Probabilities for alternative and optional rules can be specified.

```
subj = A('weather', ('forecast', 0.2), ('weather forecast', 0.2))
```

In this example the *"forecast"* and *"weather forecast"* will be selected with the probability of 0.2 and the probability of *"weather"* will be the complement probability, 0.6. This allows us to sample from more complicated subtrees with higher probability.

Building a decoder

Kaldi decoder requires both acoustic and language models. In our case the acoustic model is provided by the department. When LM is ready, Kaldi decoder can be built.

After assembling the decoder with a build script, it can be also tested within the ASDF. Statistics are computed from a test set that was created earlier from call logs when LM was built. The test set can be also tested with the Google ASR which renders a good comparison between the two recognizers.

We occasionally used CloudASR[3] for manual testing. With CloudASR, it is very easy to deploy and test Kaldi ASR. It is accessible through web interface by anyone who wants to try the decoder out by his own voice.

[3] `https://github.com/UFAL-DSG/cloud-asr`

5. Results

This chapter summarizes the results achieved with the PTINY dialogue system. We describe the 2014 MTA App Quest admission in the first part. Then we go through the subjective user satisfaction results collected from CrowdFlower. And finally we compare the subjective user satisfaction between the Google and Kaldi ASR.

5.1 App Quest 3.0

At the beginning of February 2014, we participated in the contest App Quest 3.0[1] by Metropolitan Transportation Authority (MTA)[2]. The contest rules allowed registering teams and individuals around the globe and required to submit an application that utilizes at least one of the MTA data sets or APIs and includes the ability to update the data.

We registered in the Accessibility Innovation category because the primary features and functionality of PTINY best address the end user with visual impairment. Our keyword database can be actualized any time from the server and we utilize MTA data sets, therefore PTINY is eligible to participate.

The application was, however, required to run on one of many mobile or desktop platforms. The PTINY is rather a phone service, therefore we decided to create a web page that enhances the accessibility even more.

A US number was provided by the department for the competition and three VMs were employed. We submitted PTINY[3] as an operational dialogue system, despite the fact that some features were not yet finished.

PTINY web page

The web page[4] created for the competition contains the overview of PTINY, examples of the features, terms of use and most importantly a "try it now" section shown in figure 5.1, in which a visitor has the opportunity to call PTINY directly through the web page.

We utilized webrtc2sip gateway[5] to create a *Call us Now* button. It allows any web browser supporting WebRTC protocol to call our SIP account and to try out PTINY without the need of calling a number. This includes mobile devices, too.

One additional VM was used for handling the button calls.

PTINY demonstration video

Another requirement was to provide a video link along with the submission. The video should clearly explains the features and functionality through a comprehensive demonstration. With the help of my colleague's voice, we created a video

[1] http://2014mtaappquest.challengepost.com/
[2] http://www.mta.info/
[3] http://challengepost.com/software/alex-information-about-public-transportation-in-new-yo
[4] http://alex-ptien.com/
[5] http://click2dial.org/u/index.html

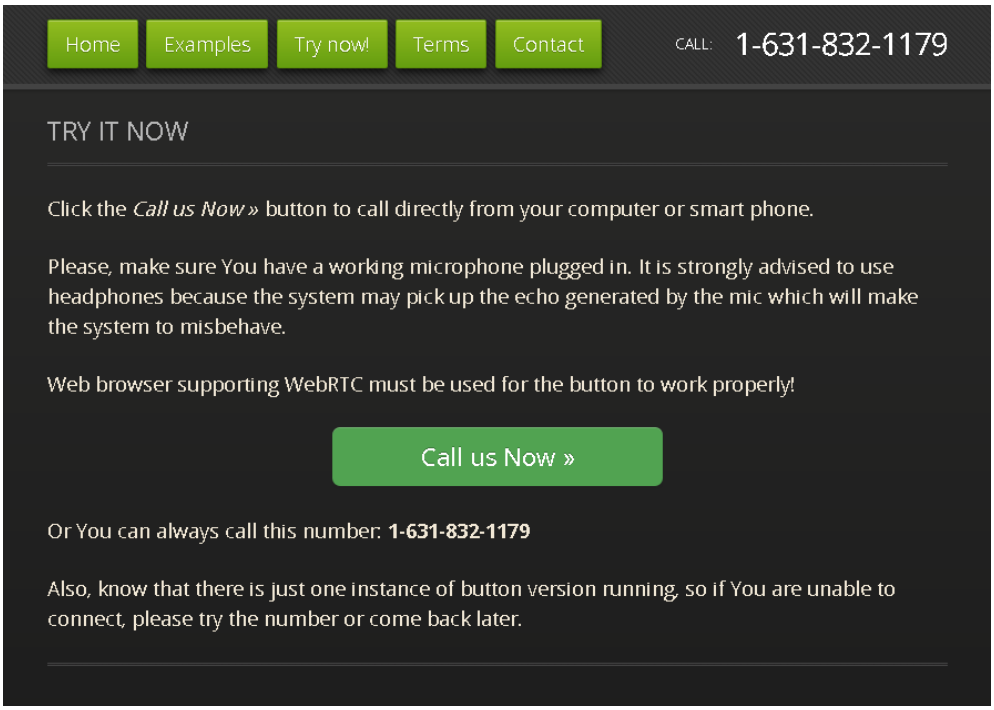


Figure 5.1: Web page with the *"Call us Now"* button for the 2014 MTA App quest.

demonstrating the features by an example call with detailed description. [6] We also elevated the fact, that it can be a great asset for the visually impaired.

Competition results

Unfortunately, PTINY was not among the winners and there was no ranking either, so we do not know how close to winning it was. Even more disappointing was the fact that we collected virtually zero calls. As the rules state, judges are not required to test the application and may choose to judge based solely on the text description or demonstration video. Our hope was that PTINY would attract at least the curiosity of some other competitors.

We know for certain that our solution scored poorly in one of the judging criteria which was utilizing MTA API. We only utilize MTA datasets.

However, the thing we cherish the most about our solution is that, while others were competing among each other within the same class of mobile applications, PTINY brought a new point of view on providing information about public transportation, with which a human can simply chat.

5.2 CrowdFlower – subjective user satisfaction

Every call job we launched on CrowdFlower had the same feedback form with questions listed in table 5.1. These questions measure the quality of every component of the dialogue system. The first question about achieving objectives evaluates the whole dialogue system, especially DM. The second question about

[6] `https://youtu.be/wtlFCJj8faE`

system phrasing measures the quality of NLG , while the third one about the voice quality is concerned with TTS. And the last question asking about how the system understood the caller evaluates the ASR and SLU components.

Have you found what you were looking for?	Yes-No question
The phrasing of the system's response was:	range of 1 to 4 from Very poor to Very good
The quality of the system's voice was:	range of 1 to 4 from Very poor to Very good
The system understood me:	range of 1 to 4 from Very poorly to Very well

Table 5.1: CrowdFlower feedback form questions with choice ranges.

The results from each CrowdFlower job provided in a CSV document were collected and joined for corresponding ASR. Even with the feedback form fields marked as mandatory, there were a few missing values in the results. Thus we collected less feedback forms than calls.

In addition to the compulsory questions evaluating the job, there was an optional general comments field. Comments gave a good overall image of the contributor satisfaction as callers could express themselves freely and in few cases, they helped enhance the system.

5.2.1 Google ASR

It is important to note that the results from CrowdFlower call jobs that contributed to the Google ASR evaluation were collected while some features of the system were not yet implemented. However, the call job always encouraged callers to address only the features and functionalities working well. Therefore the user satisfaction should not be influenced by the fact that the system changed over time.

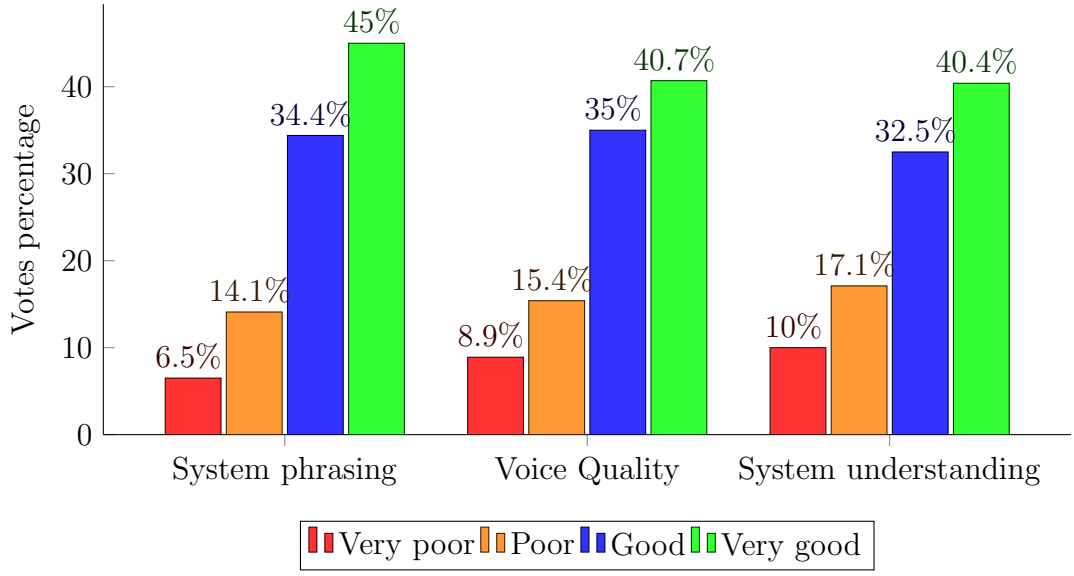


Figure 5.2: Google subjective user satisfaction histograms for questions 2-4 from table 5.1

We launched seven jobs with increasing number of ordered calls each time. In the settings, we allowed contributors to participate only once per job to collect

diverse data which caused the job to be rather unattractive, hence the collection quite slow.

Totally, we collected 369 valid feedback forms. The figure 5.2 shows histograms of questions 2, 3 and 4. It is clear that more than a half of callers were rather satisfied with the service. The first yes-no overall question is shown in figure 5.4.

In the general comment section, 101 contributors shared a mixture of positive and negative comments.

"Awful system - not working at all."
"Good service, no problems."
"It made me do it twice before it heard me."
"Excellent directions."

This short list is a sample of repetitive comments in similar vein.

5.2.2 Kaldi ASR

The same setup of CrowdFlower call jobs was used when launching the same rate of tasks as in the case of Google ASR. We have collected five jobs with 280 valid feedback forms. All of those five jobs had unique configuration urging callers to ask about different particular features and waypoints. It was the same set of configuration as in the case of Google jobs, however, two of those configurations were split into separate jobs due to the development process. This is why Google has more jobs and it only means that contributors could participate in jobs with those two configurations twice.

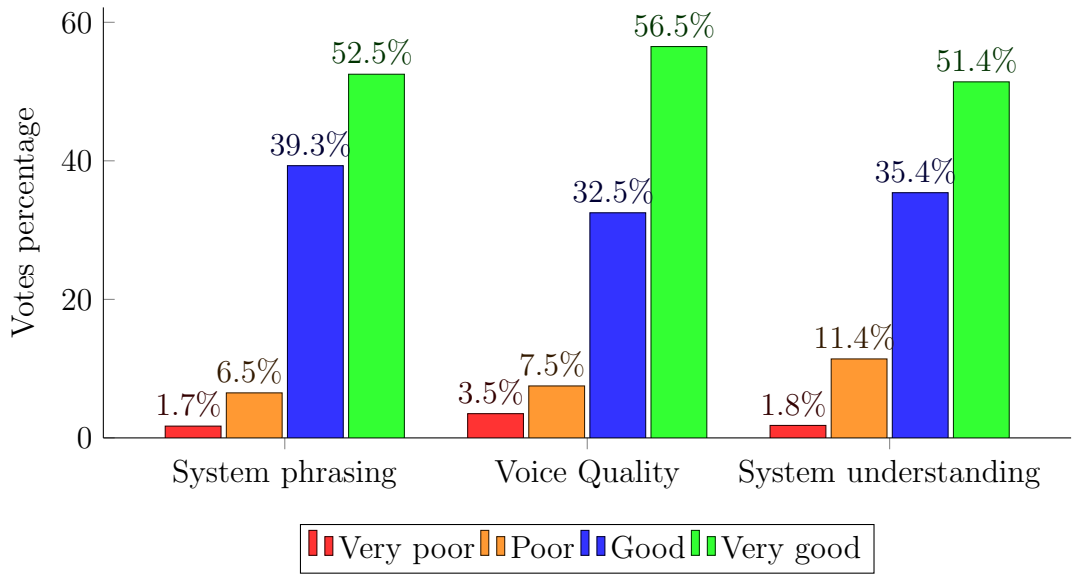


Figure 5.3: Kaldi subjective user satisfaction histograms for questions 2-4 from table 5.1

The figure 5.3 shows histograms for questions 2, 3 and 4. It is clear that very few contributors were unsatisfied with the service. It indicates an improvement in comparison to the Google ASR. The first yes-no overall question is shown in figure 5.4.

Only 60 contributors decided to write a general comment which were generally positive.

"Good, fast service."
"I liked this."
"It would be nice if the voice was more fluid. It sounds too robotic."
"Wow! An automated system that understands my needs!"

This also indicates an improvement against the Google ASR.

5.3 Comparison – summary

It is clear that the system was able to respond both with Google and Kaldi ASR. Although notably better results were achieved with Kaldi ASR as the subjective user satisfaction displayed in figure 5.4 is in favor of Kaldi ASR.

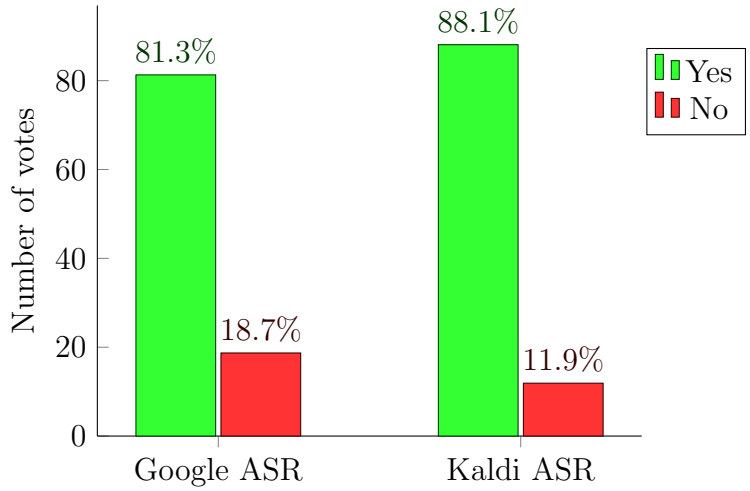


Figure 5.4: User satisfaction, have found what they were looking for

Callers communicating with PTINY using Google ASR did obtain what they were looking for in 81.3%. Whereas Kaldi ASR callers were satisfied in 88.1%. Means and standard deviances of individual questions are displayed in table 5.2. The mean of all the questions increased with Kaldi ASR while the standard deviance of each question decreased.

	Google		**Kaldi**	
	Mean	**StD**	**Mean**	**StD**
System phrasing	3.17	0.91	3.43	0.69
Voice Quality	3.07	0.95	3.42	0.78
System understanding	3.03	0.98	3.36	0.76
Overall performance	0.81	0.39	0.88	0.32

Table 5.2: Mean and standard deviation comparison of Google and Kaldi ASR

The measurement of the question number 2, evaluating the voice quality, was following the improving tendency of the other questions. The TTS component,

however, was still the same. This phenomenon could be caused by the user satisfaction from completing the job with ease. When ASR does not recognize an utterance the DM does not change the state and the user needs to repeat himself. This can be annoying and ultimately, it may be reflected on the feedback.

ASDF ASR comparison

We compared both ASR components individually, isolated them from the dialogue system. ASDF allows us to divide the transcriptions from call logs into training and testing sets along with respective audio tracks. After training LM and building Kaldi decoder, we were able to test it on unseen utterances and compare it with Google ASR.

The measure, we were concerned with, was Word Error Rate (WER). WER of Google ASR was 31.33% while WER of Kaldi ASR was 16.93%. This indicates that adapted Kaldi ASR was much better on all of 149 test utterances.

5.4 Future work

The provided PTINY solution would benefit from further SLU, NLG and static knowledge database improvements for covering another area. Utilizing MTA API would furnish real-time information about connections, for example current position of train. The route descriptions can be really extensive, therefore it would be a valuable feature to send the directions in SMS form on demand. Statistical SLU for robustness may be also profitable to develop.

Conclusion

This thesis presented a dialogue system providing Public Transport Information in New York (PTINY) developed using the Alex Spoken Dialogue Framework (ASDF). This involved creating a custom handcrafted Spoken Language Understanding (SLU), dialogue manager and a Natural Language Generator for the public transport domain. Bootstrapping sentences were used for creating the first operational system. All of the involved components were further enhanced incrementally while the system was evaluated by real users. We collected a static, easy-to-update knowledge-base from the public transit providers in New York. Additionally, the dialogue system supports weather and current time queries for the entire United States.

CrowdFlower crowdsourcing platform was utilized for collecting audio data. The collected data were later transcribed using CrowdFlower platform as well. A grammar capable of generating sentences likely to be used by the users for public transport information inquiries was created. The purpose of the grammar was to substitute the lack of data needed for creating a good Language Model (LM). With the combination of CrowdFlower and grammar data, a LM was trained and subsequently Kaldi decoder was built.

The Kaldi ASR was compared with the cloud-based Google ASR. It was shown that in a limited domain Kaldi is able to achieve notably better results than Google ASR. Aside from the comparison of both ASRs within the ASDF, feedback forms from CrowdFlower served as a subjective user satisfaction measure for the comparison of the dialogue system as a whole with different ASR components. It was shown that PTINY achieves better results with Kaldi ASR. Moreover, the dialogue system proved to be stable and beneficial in helping everyday commuters.

The goals of this thesis were successfully completed and the solution was integrated with the ASDF. We proved that the ASDF is suitable for creating spoken dialogue systems. Furthermore the PTINY was capable of competing alongside commercial applications in the 2014 MTA App Quest 3.0.

5.5 Acknowledgements

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme “Projects of Large Infrastructure for Research, Development, and Innovations” (LM2010005), is greatly appreciated. We also thank Fred Concklin for assisting with the MTA contest and to Ondřej Dušek, Ondřej Klejch, Ondřej Plátek and Lukáš Žilka for their useful comments and discussions.

Bibliography

[1] Guo, R., Zhu, X., and Hao, Y. A Chinese spoken dialog system for blind men, In: *Systems, Man, and Cybernetics 2001 IEEE International Conference on*, Vol. 2, IEEE, pp. 865–868.

[2] Jurčíček, F., Dušek, O., Plátek, O., and Žilka, L. Alex: A Statistical Dialogue Systems Framework, In: *Text, Speech and Dialogue*, Springer International Publishing, 2014, pp. 587–594.

[3] Roy, N., Pineau, J., and Thrun, S. 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.

[4] Morbini, F., Audhkhasi, K., Sagae, K., Artstein, R., Can, D., Georgiou, P., ... and Traum, D. Which ASR should I choose for my dialogue system, In: *Proceedings of the 14th annual SIGdial Meeting on Discourse and Dialogue*, 2013, pp. 394–403.

[5] Plátek, O., and Jurčíček F. Free on-line speech recogniser based on Kaldi ASR toolkit producing word posterior lattices, In: *15th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, 2014, p. 108.

[6] Plátek, Ondřej. Speech recognition using KALDI, 2014.

[7] Ipeirotis, Panagiotis G., Foster Provost, and Jing Wang. Quality management on amazon mechanical turk, In: *Proceedings of the ACM SIGKDD workshop on human computation*, ACM, 2010, pp. 64–67.

[8] Gino, F., and Staats, B. R. Samasource: give work, not aid, ‚Harvard Business School NOM Unit Case, 2012.

[9] Dušek, O., Plátek, O., Žilka, L., and Jurčíček, F. Alex: Bootstrapping a Spoken Dialogue System for a New Domain by Real Users, In: *15th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, 2014, pp. 79–83.

[10] Ma, T., and Jan-Knaap, G. Analyzing Employment Accessibility in a Multimodal Network using GTFS: A Demonstration of the Purple Line, Maryland, 2014.

[11] Biewald, Lukas. *CrowdFlower resource library* [online]. 2015 [cit May 5, 2015]. Available from: `http://www.crowdflower.com/overview`

[12] UFAL-DSG *The Alex Dialogue Systems Framework - Public Transport Information*, [online]. 2015 [cit May 5, 2015]. Available from: `http://ufal.mff.cuni.cz/alex`

List of Tables

List of Abbreviations

AM Acoustic Model . . . 4
API Application Programming Interface . . . 3
ASDF Alex Spoken Dialogue Framework . . . 2
ASR Automatic Speech Recognition . . . 2
CML CrowdFlower Markup Language . . . 19
CSV Comma Separated Values . . . 20
DA Dialogue Act . . . 9
DAI Dialogue Act Item . . . 9
DM Dialogue Manager . . . 10
GTFS General Transit Feed Specification . . . 8
LM Language Model . . . 4
MTA Metropolitan Transportation Authority . . . 2
NLG Natural Language Generation . . . 11
PTI Public Transport Information . . . 2
PTINY Public Transport Information in New York . . . 2
SIP Session Initiation Protocol . . . 4
SLU Spoken Language Understanding . . . 7
SRILM SRI Language Modeling Toolkit . . . 4
TTS Text to Speech . . . 4
UFAL Institute of Formal and Applied Linguistics . . . 3
VAD Voice Activity Detection . . . 4
VM Virtual Machine . . . 5
WER Word Error Rate . . . 29

CD contents

The compact disk included with the thesis has following structure:

- **Sources** – Folder with source files for the ASDF and PTINY. In "*Sources/alex/alex/applications/PublicTransportInfoEN/hclg/models/*" there is the Kaldi decoder located.
- **Technical_documentation.pdf** – The documentation for the maintainers of PTINY
- **User_documentation.pdf** – The documentation for the callers.
- **Thesis.pdf** – The electronic version of this thesis.