THE IBM CONVERSATIONAL TELEPHONY SYSTEM FOR FINANCIAL APPLICATIONS


IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA

ABSTRACT

We describe our development work on a telephone-based conversational system in the domain of mutual fund transactions. This system uses several components including robust large vocabulary continuous speech recognition, natural language understanding, dialog management, and text-to-speech synthesis technologies.

1. INTRODUCTION

The advent of improved speech recognition and affordable compute power has made spoken language man-machine interfaces increasingly attractive. The argument for these conversational interfaces becomes even more compelling for “displayless” telephony applications. Traditionally, telephony applications have been constrained to use cumbersome touchtone interfaces requiring many keypresses to navigate complicated menu hierarchies. Speech input allows the telephone to become the ultimate thin client, providing ubiquitous conversational access to both traditional legacyivr-based services and cutting edge web-derived content. Here we present IBM’s state of the art ViaVoice Telephony system.

2. SYSTEM ARCHITECTURE

Our conversational telephony system is a pipeline of several components communicating via a hub. This system is developed using IBM ViaVoice Telephony NLU toolkit. The overall system architecture is illustrated in Figure 1.

The first component is the telephony module that provides basic telephony services such as answering calls, playing and recording audio, collecting dtmf, echo cancellation, begin and end point speech detection and line supervision. The audio from telephony is sent to the speech recognition system. The recognized text passed to a classer which identifies simple named classes in a sentence. The classed sentence is then passed to a parser which generates the most probable semantic parse. The parse and the canonicalized class information is then passed to the form-based dialog manager. The FDM generates a text string after zero or many consultations with the backend. The text is then passed to the text-to-speech component. The audio from TTS is sent to the telephony component. We discuss the details in the rest of the paper.

3. LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION SYSTEM

The speech recognition engine is the standard IBM product level stack decoder[1].

It consists of 5 major components:

- MFCC-based front end signal processing resulting in a 39-dimensional feature vector computed every 10 msec
- Labeling of the acoustic feature vector according to the subphonetic units constructed using decision tree networks

ACKNOWLEDGMENTS: We thank the worldwide team responsible for putting together this architecture in the form of the ViaVoice Telephony NLU Toolkit with a special thanks to our business unit in West Palm Beach.
**4. STATISTICAL NATURAL LANGUAGE UNDERSTANDING**

We use two statistical parsers to extract meaning from an utterance. The first parser identifies simple named classes like 'date' or 'price', and is referred to as the "classer." The second parser identifies the remaining semantic information, and is referred to as the "parser." Both parsers are trained from a corpus of annotated sentences[3].

With this approach, two tasks are required to build a parser or classer for a new domain. First, semantic nonterminal and preterminal vocabularies must be defined. These are designed to match the basic actions and objects of the domain in a manner similar to the way a GUI designer chooses graphical components to reflect a task's actions and objects. Second, an initial corpus of training sentences is collected and treebanked with the new semantic nonterminal values. From this initial corpus we bootstrap a system, reducing the subsequent treebanking effort to simply correcting the system's errors.

We also use a "canonicalizer" to map named class expressions such as "the fifteenth of April" into a canonical form like "04/15/2000." We elaborate on each of these components below.

**Statistical Classing**

We employ a classer to identify simple named classes in a sentence. An application designer enjoys a large amount of freedom deciding which constructs to analyze in the classer and which to defer to the parser. Our rule of thumb is that classes which are easily identifiable using only local context should be handled by the classer. Examples of the classes used in our mutual fund system are:

- **ACCOUNT** - Account numbers when users sign in: "my account number is ACCOUNT( one two three four )".
- **AMOUNT** - Dollar amounts: "buy AMOUNT( three thousand dollars )".
- **FUND** - Fund names: "sell FUND( fidelity magellan )".
- **PERCENT** - Percentages: "sell PERCENT( all ) of it".
- **SHARES** - Number of shares: "sell SHARES( one hundred shares )".

We view classing as simply a particularly shallow form of semantic parsing. A sample parse tree for classing is shown in shown in figure 2. Words form the leaves of the parse tree, and each word has an associated "semantic part of speech tag" which occupies the preterminal node directly above the word. Words that are not part of a class are are tagged as
"word" indicating that the word should be passed on as-is to the semantic parser. The remaining words are tagged with the appropriate class tag and are joined together into constituents labeled by the class names. Finally, all top-level nodes are combined into a single constituent labeled by the root node label "IS!”. The text string corresponding to each of the class constituents is replaced by the class name to create a simplified English expression. This "classed" English forms the input to the semantic parser.

5. FORM-BASED DIALOG MANAGER

The form-based dialog manager (FDM) is a mixed-initiative, application-independent module that handles basic dialog steps such as

- prompting for missing information
- clarifying ambiguous information
- inheriting information from context
- confirming at critical stages of dialog
- offering context-sensitive help

The FDM is an intrinsically application-independent engine that is initialized with an application-dependent form-script defining the behavior of the dialog manager. The form-script contains a set of forms corresponding to the tasks that the application is expected to perform. More about the forms can be found in [4]. The FDM preprocesses the parse tree and class information to assemble a list of attribute-value pairs. Alternatively, this list can be assembled outside the FDM and can be passed to the FDM. The FDM processes the attribute-value pairs in context to determine its next action. The FDM may need to clarify values of some attributes (such as an underspecified fund name). This involves consulting a disambiguator module. Currently the disambiguator is a sub-module of the backend.

The form-script specifies the application-specific protocol used by the FDM to communicate with the backend. For instance, when all the necessary information to perform a task is available to the dialog manager (i.e. all the mandatory slots are filled in a form), the dialog manager sends a command string from the form to the backend via the hub. The string
encodes the command and the necessary arguments to the command. The dialog manager in turn expects an array of strings to be passed back from the backend via the hub. The actual command string syntax is up to the writers of the backend and the form script. It is clear that in each dialog turn the FDM may need to consult the backend many times: to disambiguate an attribute's values and to perform the final task of the form in case the form is complete. The FDM constructs a text message to be sent to the user based on the backend response.

6. BACKEND

The backend is the module responsible for performing the tasks in the application. For many applications, the backend will be a wrapper around a legacy application such as a database or a transaction processing facility. Since clarification of ambiguous information such as a fund name in one's current portfolio requires access to the database, we currently treat the disambiguator as a sub-module of the backend. The backend receives requests from the dialog manager as a string. It returns a return code and an array of records (strings).

7. TEXT-TO-SPEECH PROCESSING

A phrase splicing and variable substitution system is used in Text-To-Speech (TTS) processing. This system offers an intermediate form of automated speech production lying in-between the extremes of recorded utterance playback and full text-to-speech synthesis. The system incorporates a trainable speech synthesizer and an application specific set of pre-recorded phrases.

A trainable TTS system is first built via 45-60 minutes of recorded speech together with corresponding laryngograph signal. This data is used to train a set of decision-tree state-clustered HMMs. The HMMs are then used to segment the training database into decision tree leaves. A pronunciation dictionary is built to contain all of the synthesis information, such as segment duration, energy, pitch, endpoint spectral vectors, and the location of the moments of glottal closure through regions of voiced speech.

The phrases to be spliced are also recorded together with a laryngograph signal. These files are processed using the pre-trained HMMs to calculate word level and phone level alignment. A splice file dictionary is constructed to contain every word sequence of every length present in the splice files, together with the phone sequence aligned against those words.

During synthesis, the text is converted into a phone sequence. A search algorithm is used in conjunction with the splice file dictionary and a pronunciation dictionary (used to supply the pronunciations of unknown words) to determine the phone sequence to be synthesized given the text. The process creates the phone string from phone sequences seen in the splice files where possible, and a pronunciation dictionary where not. A left to right greedy search algorithm is applied to find the segment with the longest phone sequence matched in the splice files. This results in fewer joins and hence higher quality output speech. After all the possible segments are obtained, the synthesizer selects modified and concatenated segments using dynamic programming from these synthesis inventories to generate the output speech. Further details of this phrase splice system can be found in [5].

The system enables the seamless splicing of pre-recorded phrases both with other phrases and with synthetic speech. It enables very high quality speech to be produced automatically within a limited domain.

8. REFERENCES