# NLify: Third-Party Programming Support for Spoken Natural Language Interfaces

Seungyeop Han[1,2], Matthai Philipose[2], Yun-Cheng Ju[2]
[1]University of Washington [2]Microsoft Research

## ABSTRACT

This paper presents the design and implementation of a programming system that enables third-party developers to add spoken natural language (SNL) interfaces to mobile applications. Existing systems either restrict SNL capabilities to first-party applications or limit developer-defined spoken interactions to keyphrases rather than broad natural language. An examination of expert workflow reveals that the primary challenge is in gathering comprehensive sets of paraphrases for each command and in selecting and tuning corresponding statistical models for speech and language processing. We address the former problem by integrating automated statistical machine paraphrasing and webscale crowdsourcing into the developer workflow. We address the latter by developing a classifier architecture designed to be robust across app domains. We have realized our design fully as an extension to the Visual Studio IDE. Based on a new benchmark dataset with 3500 spoken instances of 27 commands from 20 subjects and a small developer study, we establish the promise of our approach and the impact of various design choices.

## 1. INTRODUCTION

Visual attention, time and free hands are often at a premium in mobile settings. Speech is therefore emerging as an attractive alternative for interacting with the phone. When speech-enabled interactions are few, *keyword-based* interfaces [4] that require users to remember precise invocations are adequate. As the number of such interactions increases, users are more likely to forget keywords, and *spoken natural language* (SNL) interfaces that allow users to express their functional intent without conforming to a rigid syntax become desirable. Prominent "first-party" systems such as Siri and Google Voice Search offer such functionality on select domains today. In this paper, we present a system that enables any ("third-party") developer to add an SNL interface to their application.

We focus on enabling command-and-control (C&C) style interactions (e.g., "When is the next 210 to San Jose?"). In this mode, the user speaks a single sentence or phrase to indicate a single *intent*, i.e., a single handler function supplied by the developer in the underlying program (e.g., `findBusTime(route,destination)`). The intent may be expressed in several different "natural" ways, e.g., "When's the 210 to San Jose expected?", or even "Next San Jose 210 bus". The role of the SNL front end is to analyze the speech and dispatch to the correct handler, with parameters (also known as *slots*) correctly instantiated. More sophisticated functionality, such as support for dialog, composition of intents and extended dictation, is useful but substantially more challenging. However, although single-intent C&C is simpler, it is still broadly useful (in fact it is the mode supported by today's first party systems), hard for non-experts to implement and the corresponding third-party programming problem is unaddressed.

| | | |
|---|---|---|
| show me the current time | what time it is now | what time is it now |
| what is the time | show current time | what time is it currently |
| time | what time please | check time |
| what is the current time | show time | the time now |
| may i know the time please | what is the time now | tell me the current time |
| give time | current time please | what's time |
| show me the time | say the time | time now |
| show me the clock | find the current time please | tell me the time |
| tell me what time it is | what time is it | can you please tell me what time it is |
| what is time | what is current time | tell me current time |
| current time | what time is it tell me | give me the time |
| tell what time it is | time current | time please |
| list the time | what's the time | show me the time now |
| what time | tell current time | |

**Figure 1: Forty-two ways to ask for the time.**

Although speech *recognition* is moderately robust today, and available via standard APIs on most platforms, *understanding* natural speech is still challenging. Figure 1 captures the essential problem, based on spoken data collected for this paper: even a request for the time may be *paraphrased* in dozens of ways (the figure samples 42). For commands such as checking for messages from Facebook, we have observed over 200 plausible paraphrases. Second, noise introduced by speech recognition engines introduces further variability (routinely over 20%) in the set of commands to be understood by the natural language engine, thus adding to the challenge of recognition.

Unlike in speech recognition, no off-the-shelf solutions exist yet to handle this variability. Classifiers for natural language (even for C&C interactions) are still de-

veloped on a domain-by-domain basis, with experts sequencing through data collection, feature selection, statistical model selection, training and tuning. Tuning often involves optimizing the interaction between the speech module and the language-understanding module, ensuring acceptable resource usage including partitioning computation between client and cloud, configuring "garbage" models to capture irrelevant speech and incorporating dynamically available information relevant to classification.

Following the expert workflow to produce even moderate quality natural language recognition systems would be challenging for most developers. On the other hand, even relatively simple applications (e.g., turning a night-light app on/off) can benefit from SNL enablement. We present the NLify system, aimed at bridging this gap between utility and ease of implementation. We capture the above workflow in a development environment, automate as many steps as possible, and provide structured support so developers can easily complete the remaining few steps. We demonstrate that although designing general SNL interfaces may require a variety of expert choices, at least in the case of C&C interfaces, it is possible to distill these choices into a single system arguably usable by non-expert programmers to achieve useful levels of performance. We target ease of development comparable to that of developing GUIs.

Our specific contributions are:

- The design and implementation of a system for programming third-party SNL interfaces. Innovations include the integration of crowdsourcing and automated machine paraphrasing (MP) services into the programming workflow, techniques for converting developer examples into a statistical language model that accommodates both data sparsity and garbage inputs, and a variation on standard vector-space models to better classify parsed sentences to intents.
- A new and extensive benchmark dataset including roughly 3500 spoken utterances from 20 subjects targeted at 27 different intents, 1600 paraphrases for these intents produced by crowdsourcing and 848 paraphrases produced by a MP system.
- A quantitative evaluation of our implementation showing overall recognition rats of 85/81% across intents/slots, along with drill-down analysis of the impact of our design decisions.
- A small qualitative study of programmers gauging the usability and utility of our system.

To the best of our knowledge, this is the the first proposal for a programming model to support third-party development of spoken natural language interfaces for mobile devices.

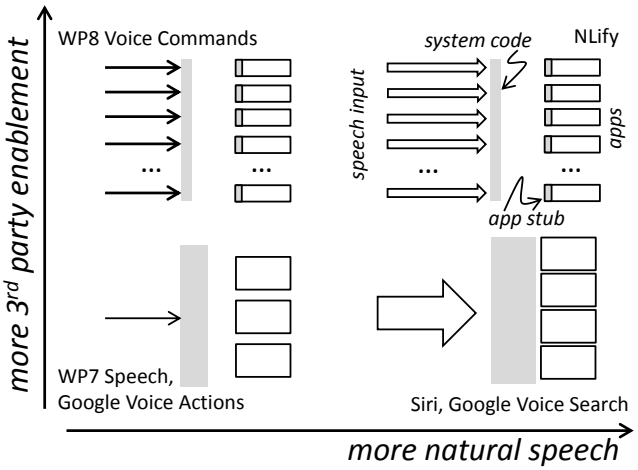



**Figure 2: Design options for mobile speech.**

## 2. ARCHITECTURAL SPACE

Several production systems allow for speech interaction with mobile devices. Figure 2 organizes the main options by the degree to which they allow natural language (versus some form or keyword of keyphrase) and allow developers speech-based access to apps. The figure shows four main classes of designs.

At the bottom left, the earliest and simplest options (e.g., Microsoft's Windows Phone 7 speech and Google's Voice Actions) support select functionality (drawn as a few large rectangles) such as searching the web, making and responding to phone calls and SMSes, launching apps, playing music and accessing the calendar. Users are required to use fixed keywords or phrases, e.g., "Search for X" or "Call X", drawn as a single (defined by one entity) narrow (rigid syntax) arrow. All speech functionality is completely implemented in the the system-provided interface (drawn as a moderately thick gray vertical bar). When the number of commands targeted is small, anecdotal evidence suggests that such keyword-based systems are usable.

Systems such as Siri and Google Voice Search (bottom right) support a much larger set of commands. In fact, given that they provide a front end to generic question-and-answer systems such as Wolfram Alpha and Google Search, the number of commands they support is unbounded. In these cases, the users mental model is to "say anything" (single broad arrow) to the system, which represents a digital valet or an all-purpose search system. The speech interface is heavily engineered by the platform provider (thick gray bar) to provide excellent performance with a few selected applications (few boxes). Both the iOS and the Android system boast extensive app ecosystems, however, and this scheme denies spoken natural language support to the overwhelming majority of these apps.

Windows Phone 8 (WP8) Voice Commands [4] suggests one approach to scaling. Applications are allowed

```
<CommandPrefix>Magic Memo</CommandPrefix>
<Example>enter a new memo</Example>
<Command Name="newMemo">
<Example>enter a new memo</Example>
<ListenFor>Enter [a] [new] memo</ListenFor>
<ListenFor>Make [a] [new] memo</ListenFor>
<ListenFor>Start [a] [new] memo</ListenFor>
<Feedback>Entering a new memo</Feedback>
<Navigate /> <!-- Defaults to Main page -->
</Command>
...
```

**Figure 3: Third-party speech extension in WP8.**

to provide patterns of spoken text to dispatch on, along with the handler functions to dispatch to (drawn as many small boxes, each with a gray "app stub" representing patterns and handlers to be provided by developers). The system focuses on exact matching against application-provided keyphrases and dispatching (thin gray bar) to handlers.

Figure 3 shows a concrete example of how a memo app may add a speech interface [4] to a command, in this case a command to open a new memo. The developer provides expected spoken variants of the command using the `<ListenFor>` tag and by calling out optional words explicitly. The handler is a target app page denoted by the `<Navigate>` tag, here left empty to default to the main page. Finally all commands to this application are expected to be prefixed by the command prefix `Magic Memo`. The API allows the app to register these commands with the OS dispatcher, which given the keyphrase followed by sequence of words, will dispatch to the appropriate app page.

Although developers are allowed to list a variety of deterministic matches for a given handler, no special support is provided to handle the large variation in typical spoken speech due to redundancies in natural language syntax, peculiarities of spoken language and noise in speech recognition. As such, users will typically be able to use a larger variety of keyphrases (thicker arrows), but most developers will fall short of supporting "natural-language scale" variety that Figure 1 touches on. On the other hand (as illustrated by multiple arrows), users are no longer encouraged to "say anything". Their mental model is one of "natural-language shortcuts" to the particular applications they installed. Although it promises less, we believe this metaphor has a combination of utility and tractability that makes it competitive with the valet or search metaphors.

In principle, if WP8 developers specify enough variants of a command, deterministic matching against all the variants could yield an experience indistinguishable from "natural language" matching. In practice, however, due to redundancies of natural language, peculiarities of spoken speech and noise in speech front-end results, expecting developers to list most (or even most common) variants is impractical. More fundamentally, some variants are less likely than others. What is needed, therefore is a way to help developers *amplify* the examples they have in mind (i.e., to help collect relevant data) and to support *statistical* rather than deterministic matching while requiring minimal statistical sophistication from developers.

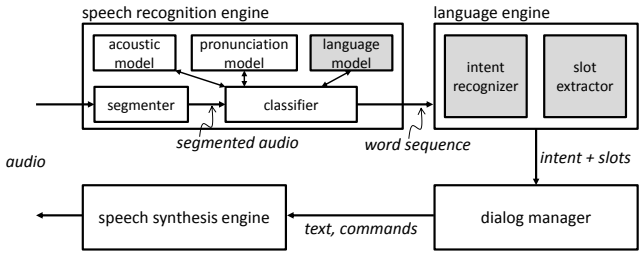



**Figure 4: Standard spoken natural language pipeline. NLify focuses on the gray boxes.**

## 3. THE SNL PIPELINE

Figure 4 illustrates a standard full-featured spoken natural language (SNL) pipeline for inferring intent from audio signals. Audio signals (sampled at 16–48kHz and 8–16 bits) are segmented. Segments are converted into sequences of words by the speech recognition engine. Each sequence is classified to produce an intent (the identity of the handler function in our setting) and its elements are classified to extract slot values (the argument values for the handler). The sequence, intents, slot values and associated confidence values (not shown in the figure) are passed onto a dialog manager, which may choose a follow-up spoken interaction with the user. For instance, the interaction may ask the user to clarify the value of a particular slot value. The text to be spoken and settings for the quality of the related speech are passed onto a speech synthesis engine. The synthesis engine renders the follow-up speech.

NLify focuses on the prefix of the pipeline that infers intent and slots from audio. Speech synthesis engines are available on all mobile platforms today, and NLify assumes their availability. NLify mirrors production mobile spoken C&C systems in foregoing sophisticated dialog management: dialog managers are just making the transition from research to production systems [19].

### 3.1 Speech Recognition Engine

Given streaming audio, the speech recognition (SR) engine first segments it using heuristics ranging from periods of silence to button presses. In NLify, we assume some segmentation scheme already exists, with speak-on-button-press followed by terminate-speech-on-silence our standard implementation.

In the process of changing raw audio to words, the

SR engine sequentially converts it into a sequence of derived low-level features (e.g., mel-frequency cepstral coefficients, MFCCs) followed by phones (or phonemes, or triples of phones called *triphones*), which can be thought of as the primitive units of the sounds produced in a particular language, followed by words. The engine is parameterized by three models in this process, and customization of the SR engine overwhelmingly happens by providing custom models.

#### 3.1.1 Acoustic Model

The *acoustic model (AM)* captures the joint distribution over feature vectors derived from raw audio and phones. AMs are typically specific to language, microphone model, device configuration, usage (e.g., background noise patterns) and broad user characteristics (e.g., geographical origin and gender). Except for applications with peculiar audio characteristics (e.g. those for functioning in an automobile), the AM is not expected to vary substantially across applications. NLify therefore assumes that an adequate AM exists on the mobile device and provides no special support for managing AMs.

#### 3.1.2 Pronunciation Model

The *pronunciation model (PM)* captures the mapping between phones and words, along with post-lexical rules on how pronunciation (i.e., phone sequences) may change in the context of multiple words. For the most part, the PM does not change with application. Current production speech libraries provide fairly comprehensive support for pronunciation across many languages.

An important relevant exception is that slot values sometimes contain proper nouns that depend heavily on the application, and the pronunciation of many proper nouns is not covered by standard pronunciation models. More broadly, individuals vary in pronouncing proper nouns, necessitating custom PMs. Platform support for augmenting the PM for such slots and values is likely important for scaling the development of these apps, but NLify does not provide special support for it.

#### 3.1.3 Language Model

The *language model (LM)* models the distribution of word sequences input to the SR engine. Deterministic models (conventionally representing using context free grammars, CFGs), simply represent the set of all allowed sentences or sequences in any single query segment. Statistical language models (SLMs) represent distributions over these sequences. The standard representation is is the $n$-gram (with $n = 1$, 2 and 3, also called the unigram, bigram and trigram models) [10] which represent the probability that various single words, length-2 sequences or length-3 sequences of words will appear in the query sequence. SLMs are known to be more robust to variation in phrasing than deterministic models, given enough training data [14]. NLify therefore uses a trigram-based SLM.

Of the three models parameterizing the recognition engine, the LM most affects per-application SNL recognition performance. For example, the distribution of word sequences to control an alarm application is very different than that for making a phone call, and both these are far smaller than the LM required for the language as a whole. When searching for the most likely sequence of words, using a custom LM for a set of expected commands (even if hundreds of commands are allowed) often yields a far smaller search space than if a whole-language (or "large vocabulary") recognizer were used. In the case that a custom model that captures an application-relevant subset of words and sequences is used, it is usually necessary to develop a *garbage model* that absorbs the remaining words and sequences. Even when a segmentation mechanism can accurately determine when a user is talking to the phone, these utterances may be out-of-grammar for many reasons [13]. The garbage model ensures that such utterances are not simply mapped onto the most similar-sounding invocabulary command in the application LM.

Alternately, users may seek to avoid the overhead of customization by using a "large vocabulary" (LV) recognizer, which is capable of recognizing arbitrary words in the language, to process incoming speech [16]. LV recognizers use models with large memory footprint and substantial computation per query, and are therefore typically accessed over the network on remote servers. For very good recognition performance, however, even these recognizers need to be augmented by the language models encompassing the particular SNL-enabled applications on a user's device. First-party domains in production are almost certainly augmented in this manner in, but no API or even proposed architecture exists for augmenting with third-party-defined domains.

The AM, PM and LM are usually combined in a joint temporal statistical model, commonly a Hidden Markov Model (HMM) [10]. Given a sequence of featurized audio frames, the HMM returns the most likely corresponding sequence of words.

### 3.2 Language Engine

The language engine (LE) converts sequences of words into a *meaning representation language* that captures the intent of the words. Although many conceivable settings require complex representation languages and command semantics, in order to facilitate robust automated processing, NLify restricts itself to the simplest commonly useful case. Every sequence is expected to represent a single sentence, and each sentence represents exactly one intent. In particular, for the C&C invocations we target, the representation language is

simply a tuple containing the function name that is the handler for the command and the slot values that constitute the parameters of the command. For example, the language engine may convert [ "when", "is", "the", "next", "210", "to", "San Jose"] to (intent = findBusTime, slot-route = 210, slot-destination = "San Jose").

The LE addresses two classification problems. Given the sequence of words, the *intent recognizer* infers the intent corresponding to the sequence. The *slot extractor* recognizes and extracts parts of the sequence corresponding to the slots for the intent. The design space for these classification problems is mostly determined by four related choices, although very many approaches have been proposed [9]. First, which classifier should be used? Second, how should the incoming word sequence by preprocessed (or "featurized") before being input to these classifiers? Third, how should slot values be parsed? And fourth, how is the training data for classifiers to be generated?

There is some consensus on the first two questions [9]. For slot extraction, statistical models that examine up to three words around the point to be classified provide the best performance, with Conditional Random Fields (CRFs) leading the others by 1-5%. CRFs, especially when operating in a streaming or "filtering" mode may then be thought of classifying each word based on the classification of the previous word and on the identity of the past three words. The intent classification problem for commands is a simple instance of the general automated text categorization problem for which many approaches exist [17]: given a sequence of words that constitute a "document", infer the class that the document belongs to. A standard representation of the space of documents, which we adopt, is as a vector space over term-frequency inverse-document-frequency (TF-IDF) scores [15]. Classification in this space may be performed using most standard classifiers [7], and we use a simple nearest-neighbor approach.

The third question is of practical importance. For example, although our example above assumed that "San Jose" was passed in as a single "word" by the SR engine to the LE, it is entirely possible that it is passed in as two words ("San", "Jose"), and it is the job of the LE to determine that the two words constitute a single attribute value. The determination may be made using statistical techniques from example data, but in domains with large numbers of slot values, the examples often do not cover many slot values. It is common therefore to have deterministic rules that identify both domain-specific strings (like organizational names) and common generic values (like numbers and times). The best solutions use a combination of statistical inference and user-provided rules to identify the values, but slot classifiers must at the very least be able to coexist with developer-specified rules, a requirement that

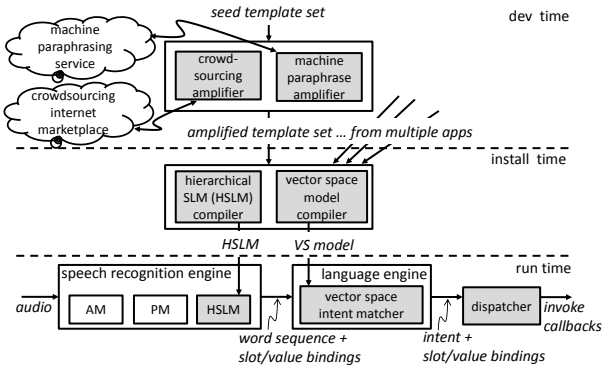



**Figure 5: NLify system architecture.**

NLify meets.

Classifier performance depends directly on training data. In the case of NLify, training data corresponds to the variety of phrases that may be used to invoke a particular command. Generating such paraphrases is a long-running problem in natural-language processing.

The gold standard for paraphrasing is to use internet-based crowds, which may be used to verify that one sentence is the paraphrase of the other [6], to generate new paraphrases of a given phrase [5] and to generate paraphrases for a common semantic form [18]. In every case, crowdsourcing has been used as an *offline* technique to acquire a corpus collected by experts for benchmarking. NLify pulls in crowdsourcing as an *online* technique to be used as part of the development cycles by developers. In doing so, we need to ensure that known issues of timeliness, quality and cost of crowdsourced results are compatible with the development process.

The lack of a large paraphrase corpora has hindered the development of automated paraphrase technologies. However, exploiting the relative abundance of bilingual corpora, recent work [2, 11] has shown promising results. These results lag human paraphrasers considerably, but given that automated paraphrasing could be much faster in time and money than crowdsourcing, we integrate a web-based paraphrase service [1] in NLify.

# 4. SYSTEM DESIGN & IMPLEMENTATION

Figure 5 shows how NLify is structured. NLify acts in three phases. When apps are being developed (*dev time* in the figure), given a small set of seed examples of utterances (called "templates") and corresponding handler invocations for a given command from the developer, NLify uses cloud-based services to amplify the set into a large, comprehensive set that the developer can select from. When an app with NLify support is installed on a phone (*install time*), the templates for that app are pooled with those from other NLified apps on the phone to derive a language model and an intent

matcher. The templates essentially act as a pattern on which statistical dispatch is performed. These components parameterize the speech engine on the phone and the NLify-provided language engine as described in section 3. At *run time*, an NLify system dispatcher listens to incoming audio, sequentially performs speech recognition and language recognition on it, and dispatches to registered handlers for templates that match.

## 4.1 Template definition language

NLify provides a simple two-level grammar definition language to define *template pattern sets* (TPSs) to match against. TPSs have the following form:
$T \in \text{TPS} \leftarrow h(s_1, \ldots, s_m) \quad d \quad t_1, \ldots, t_n \quad g_1, \ldots, g_k$
$h \in$ handler function names, $s \in$ slot names
$d \in$ description of scenario for command
$t \in$ templates $\leftarrow \tau_1 \ldots \tau_n$
$\tau \in$ tokens $\leftarrow w | s_{[r]}, \quad w \in$ word
$g \in$ deterministic grammars $\leftarrow s ::= \ldots$
$r \in$ post processing function

Each TPS is intended to correspond to one command in the app being NLified. The TPS specifies a handler function with slot variables, a natural-language description of the scenario in which the command occurs, a set of templates and a set of definitions of deterministic grammars. Each template corresponds to a distinct phrasing to invoke the command with. In addition to words, a template may contain non-terminals. The *slot tokens* must be the root of a deterministic context-free grammar $g$. Slot tokens may optionally designate a post-processing function $r$. For convenience, we allow grammars and post-processing functions to be imported from external libraries (not shown).

A concrete TPS for a stock-price checking application:

```
handleStockCheck(@companyName,@day):
"You want to know stock price of @companyName @day"
"Tell me the stock price of @companyName @day."
"How much was @companyName @day?"
"What's a share of @companyName @day?"
day ::= "yesterday" | "today"
```

The contents are similar to that for defining C&C speech interfaces in WP8 (Figure 3) with four differences: commands do not need to be prefixed by application name, variants with missing words do not have to be explicitly indicated, a description of the scenario in which the command will be invoked is required, and some non-terminals (e.g., `companyName` in this example) may be defined externally to the specification.

## 4.2 Template Set Amplification

Given the seed template set for a command from the developer, NLify provides support for expanding the set to include many paraphrases of the seed set. The developer may invoke the support either through API functions or interactively through a graphical interface. The developer is then allowed to select the results they wish to keep for their application. We choose this semi-automated approach because both the crowd and the machine are not fully reliable: bad actors in the crowd may add unacceptable paraphrases, and machine paraphrasing results are of uneven quality.

### *4.2.1 Crowdsourced Paraphrasing*

NLify requires the user to register via web browser at the crowdsourcing service and acquire an API key. Given the key and a selection of templates to paraphrase, NLify automatically interacts with the service to acquire paraphrases.

NLify provides a generic paraphrasing-task description to each worker. The description describes the general paraphrasing task with an example and (based on our experience) provides several tips on what constitutes a good paraphrases. For instance, it requires the result to be a single sentence, to use all slot values in the input sentence, to avoid changing the meaning of the sentence and to restrict responses to sentences that users may plausibly use in talking to a phone, rather than exotic but valid paraphrases.

Structuring the paraphrasing task to minimize bias is an important consideration. For instance, asking for paraphrases of "what is the time?" may cause workers to overlook imperative forms of the command "tell me the time!" or synonymous forms "what does the clock say?". Previous work has considered using videos [5], images [8], textual descriptions of the scenario or simple lists of examples [18]. The impact of these techniques in reducing bias is unclear, however. To minimize developer effort, NLify simply lets the developer select a subset of seed templates $T$ from the TPS as examples, along with the number of results or budget for the task. It presents the scenario description $d$ from the TPS and these example templates $T$ to the crowd worker with every slot instantiated with an example value. For example, for the stock application above, a task may read (`[]`s indicate values that must not be changed):

```
Scenario:
You want to know the stock price of Facebook yesterday
Example 1:  Tell me the stock price of [Facebook] [yesterday].
Example 2:  How much was [Facebook] stock [yesterday]?
Enter five other questions you may ask in this scenario.
```

We present the results to the developer after re-instating placeholders for slots, removing duplicates (after normalizing spelling and punctuation), rejecting answers that do not match the number of slots of the input template and workers who are outliers in task completion time. The developer may then choose which of the results to include in their app. The developer is free to fine-tune the task via the crowdsourcing service's web interface.

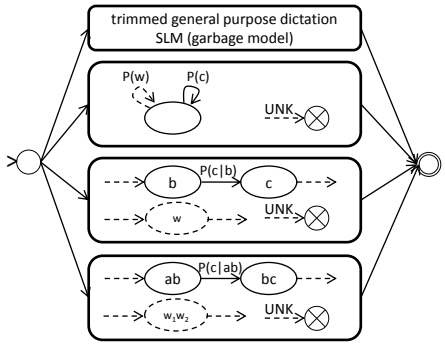



**Figure 6: Parallel n-gram + garbage model.**

#### 4.2.2 Machine Paraphrasing

NLify uses the Microsoft Contextual Thesaurus (MCT) [1] for automated paraphrasing. MCT requires as input the phrase to be paraphrased, along with the number of candidate paraphrases to be returned. If requested by the developer, NLify feeds in the seed template set $T$ to be amplified, with each seed submitted separately. All named entities in the seed are escaped as required by MCT. Results are again de-duplicated after normalizing spelling and punctuation before returning to the developer. Unlike the crowd, the MCT returns results immediately.

### 4.3 Compiling models

Given a set $T$ of amplified templates, NLify compiles then into a statistical language model (SLM) and a vector-space model.

#### 4.3.1 Compiling the SLM

NLify relies primarily on a trigram language model. A trigram model is simply a table that lists the probability $p_{ijk} = P(w_k|w_iw_j)$ that word $w_k$ may follow words $w_i$ and $w_j$ in a command. When all three words are seen in the training set, $p_{ijk}$ can simply be estimated by counting. When they are not, sophisticated techniques for interpolating these probabilities from bigram and unigram probabilities have been proposed [10].

Given that commands are short, simple and well covered by templates, NLify takes a simpler approach. It runs trigram, bigram and unigram models in parallel (Figure 6). If any model encounters an n-gram not seen in its training set (denoted `UNK` in the figure), the corresponding layer goes into a stop state. A *garbage* SLM generated from a medium-sized corpus independent of $T$ is further run in parallel with the n-grams. When all input is processed the more likely of the garbage model and the lowermost un-stopped SLM in the stack is selected as the output of the whole stack.

Key details of the structure of the SLM:

1. Slots $s$ in templates are processed uninstantiated during SLM compilation. Thus, to use the notation of Section 4.1, NLify tracks probabilities $P(\tau_i|\tau_j\tau_k)$ over *tokens*, not words. In particular, it does not track if certain instances of slots are more likely to occur with adjacent words than others (e.g., "sail to Hawaii" versus "sail to Hungary").

2. To reduce conflicts with templates in $T$, we perform simple pruning of the garbage model by removing all words that appear in $T$. Given that we expect command vocabulary to be much smaller than dictation vocabulary, we expect this crude technique to be adequate: most of the garbage SLM remains unchanged after pruning.

3. During matching, slots $s$ in templates trigger *deterministic processing* using the associated grammar $g$. Ideally, this next level of processing would also be statistical. This limitation of NLify implies that slot values will be less robust to variation in phrasing than commands in general. Our assumption is that in most cases users do not expect to be able to say slot values (e.g., times, numbers, dates, names, etc) in many different ways.

The above design has two key implications. First, since slot values are processed deterministically, they are inappropriate for capturing free-form dictation. For instance, "Make a note that ... [arbitrary note]" is disallowed, although dictation mode may certainly *follow* a command. Second, NLify moves slot extraction, traditionally a role of the language engine, into the speech engine. In fact, the local n-gram processing performed by the SLM engine to infer the identity of slots is not dissimilar from that performed by the n-ary potential functions of CRF implementations. Further, the user-defined deterministic rules are similar to those provided in language engines (see Section 3.2). However, in general, CRFs can be very sophisticated in the number and dependencies captured in their features and unlike the speech engine, they can operate in *smoothing* mode where they have the complete sentence before analysis. NLify therefore risks some performance loss.

#### 4.3.2 Compiling the Vector Space Model

NLify treats the intent recognition problem as a document matching problem. Given a spoken command (converted into a sequence of tokens by the SR engine), we seek to find the template(s) (again sequences of tokens) in $T$ (the set of all amplified templates across all TPSs) that it "best matches". The target intent is simply the one associated with the best matching template. We use a standard vector-space model [15] with Term-Frequency Inverse-Document-Frequency (TF-IDF) weighting. Relevant details follow, where we refer generically to all sequences of tokens as "templates".

We represent template $t_i$ as a feature tuple $\hat{\lambda_i} = (\lambda_{1i}, \ldots, \lambda_{Ni})$ of weights, with one weight for each of

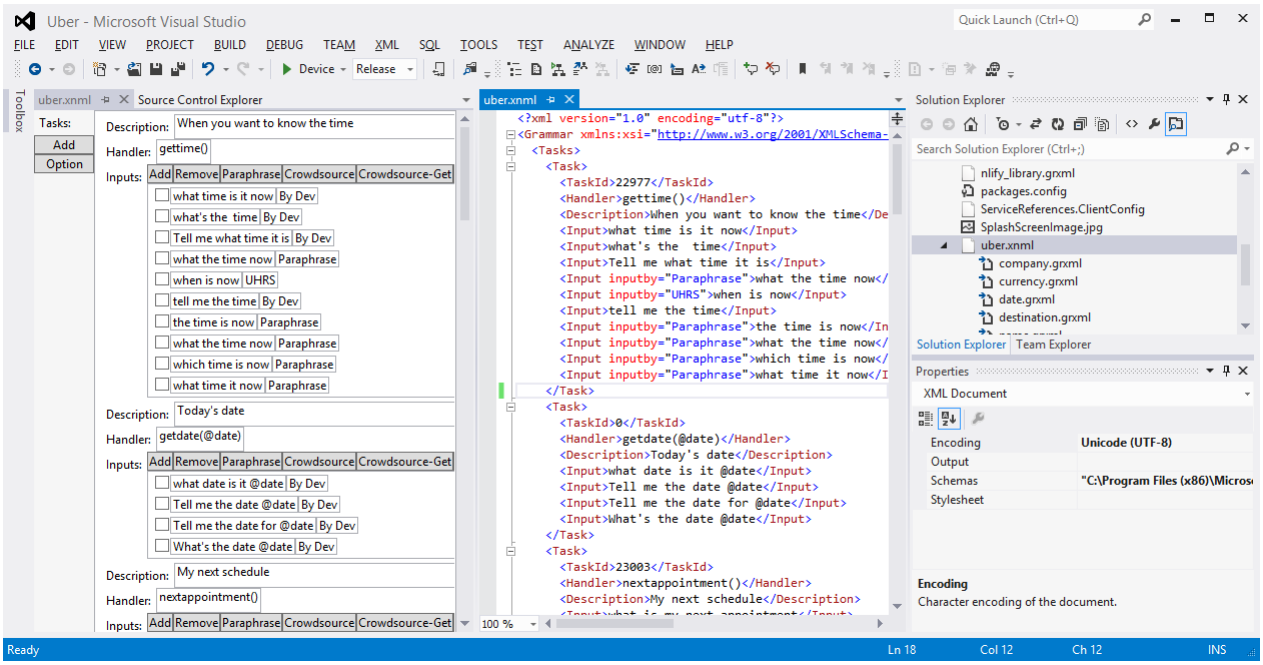



**Figure 7: IDE extension for NLify.**

$N$ tokens in $T$. The distance between templates $t_i$ and $t_j$ is the cosine of the angle between them, $\frac{\hat{\lambda_i}\hat{\lambda_j}}{\|\hat{\lambda_i}\|\|\hat{\lambda_j}\|}$. The best matching template to a query template is the closest one. The TF-IDF weight $\lambda_{ij}$ of token $\tau_i$ in template $j$ is $f_{ij} \log \frac{|T|}{|\{t \in T | \tau_i \in t\}|}$, where $f_{ij}$ is the frequency of token $i$ in template $j$. This definition of $\lambda$ is known to reward words that appear often in the document being featurized (high $f$) while suppressing those that appear in many other documents.

This (standard) formulation of TF-IDF is potentially problematic for template matching. It attempts to distinguish between individual templates, whereas we seek to only distinguish between those corresponding to different intents. For instance, all slots will get a low IDF score since they appear in *every* template for a given intent, even if they do not appear in templates with other intents and so should serve as good discriminators.

We therefore consider a *discounted* IDF score that merges all templates derived from one TPS into a single "super-template". The denominator of the IDF calculation is $|\{T_j \subseteq T | \tau_i \in T_j\}|$, where $T_j = \{t | t \in \text{TPS}_j\}$.

## 4.4 Implementation

NLify is implemented as an extension to Microsoft Visual Studio 2012. Figure 7 is a screen shot of the extension being used to NLify a calendar application. The pane provides a graphical view for defining TPSs. Three TPSs are visible, for querying time, date and schedule. `Description`, `Handler` and `Inputs` fields are visible for two of the TPSs. Inputs are the same as templates. The developer may select a seed template set using the check box to the left of each template. Pressing the `Paraphrase` and `Crowdsource` buttons will then automatically amplify the existing set of templates. The middle pane shows an alternate (XML-based text) view of the TPS list. The right pane shows some of the grammars (e.g., `date.grxml`) generated by NLify from the user's specification.

NLify uses the publicly available Microsoft Speech Recognition API for Windows Phone 8 (SAPI) as its speech engine. NLify's grammar parallels SAPI's interface. SAPI allows statistical language models to be defined as graphs with embedded grammars for rule processing. More importantly, the WP8 engine can process language models that include thousands of templates in real time on the phone. Our TF-IDF implementation is optimized to use table lookups and integer calculations to implement the log and square-root calculations. *NLify is thus implemented to function purely on-client, with no network connectivity.* We believe that as long as resource usage is modest, on-client execution is potentially an important advantage.

NLify-based applications been working on Windows Phone for several months now with over a dozen applications NLified. Due to limitations in access to the OS, the NLify dispatcher is not implemented as an OS service on the phones. Our experiments use a user-space dispatcher which requires client applications to be linked into a single application.

# 5. EVALUATION

| Domain | # | Intent and Slots | Example |
|---|---|---|---|
| Clock | 1 | `FindTime()` | "What's the time?" |
| | 2 | `FindDate(Day)` | "What's the date today?" |
| | 3 | `SetTimer(Duration)` | "Set a timer for 35 minutes" |
| Calendar | 4 | `ScheduleMeeting(Person,Day,Location)` | "Set up a meeting with Bob Brown for tomorrow in his office" |
| | 5 | `CheckNextMeeting()` | "What's my next meeting?" |
| | 6 | `ReserveRoom(NumPeople,Duration,Day)` | "Book a meeting room for 5 people for an hour on Wednesday" |
| Current conditions | 7 | `FindWeather(Day)` | "What's the weather tomorrow?" |
| | 8 | `FindNextBus(Route,Destination)` | "When is next 20 to Renton?" |
| | 9 | `FindTravelTime(Destination)` | "What's travel time to Renton?" |
| | 10 | `ConvertCurrency(Amount,SrcCurr,TgtCurr)` | "How many dollars is 17 euros?" |
| Finances | 11 | `FindStockPrice(CompanyName)` | "How much is Starbucks stock?" |
| | 12 | `RecordSpending(Money,CompanyName)` | "Remember spending 68 dollars at Ikea" |
| | 13 | `CalculateTip(Money,NumPeople,Rate)` | "Split 32 dollars and 10 cents five ways" |
| Contacts | 14 | `FindOfficeLocation(Person)` | "Where is Janet Smith's office?" |
| | 15 | `FindManagerName(Person)` | "Who is Janet Smith's boss?" |
| | 16 | `FindGroup(Person)` | "Which group does Janet Smith work in?" |
| Unit Conversion | 17 | `ConvertUnits(SrcUnit,TgtUnit)` | "How many teaspoons in a tablespoon?" |
| | 18 | `ConvertNumUnits(Amount,SrcUnit,TgtUnit)` | "How many ounces is 7 pounds?" |
| Music | 19 | `MoveToPlaylist(Playlist)` | "Put this song on the Dance playlist" |
| | 20 | `ShareWithFriend(Person)` | "Share this with Bob Brown" |
| | 21 | `IdentifyMusic()` | "What's this song?" |
| Social Media | 22 | `PostToSocial(Channel)` | "Send this to Facebook" |
| | 23 | `ListSocial(Person,Channel)` | "Any messages on Facebook from Bob Brown?" |
| | 24 | `LikeOnFacebook(Liketarget)` | "Like Toshi's Teriyaki on Facebook" |
| Local search | 25 | `FindMovie(MovieName)` | "Where is Avatar playing?" |
| | 26 | `FindRestaurant(Cuisine,Money)` | "Find me a Japanese restaurant under 20 bucks" |
| | 27 | `FindDistance(Destination)` | "How far is it to Renton?" |

**Table 1: NLify Command and Control Dataset Summary**

We seek to answer the following questions. 1. How well does do systems built using NLify perform overall? 2. How does performance scale with the number of NLified commands installed? 3. How do our design decisions impact recognition rates? 4. How does the on-phone implementation perform? What is the resource consumption of NLify? 5. What is the developer experience using NLify SDK? It the SDK usable and useful?

## 5.1 The NLify C&C Dataset

To our knowledge, no public dataset designed to evaluate mobile spoken natural language systems exists. We have therefore collected an extensive dataset that we intend to make public (Table 1). We identified 27 pieces of functionality (each corresponding to an intent) a user may access from the phone via spoken language, divided cross nine application domains. We tried to sample multiple intents in a single domain, both since these often go together in the real world, and they tend to have common vocabulary and slots as a result (which make them challenging to distinguish). For each intent, we identified plausible slot values. The number of distinct slot values were 2680 for CompanyName, Destination (271), MovieName (247), SrcCurr/TgtCurr (163), person (102), playlist (15), TgtUnit/SrcUnit (17), Day (10), Cuisine (9), Location (6), Channel (3). Remaining slot (e.g., Amount) were numbers, time durations, or currencies of unbounded cardinality implemented as external rules.

The overall goal of this dataset is to examine how well audio signals representing spoken commands can be identified given various kinds of training data. Accordingly, we collected three kinds of data.

### *5.1.1 Voice Data*

For each intent, we selected one or more slot values. For instance, we selected `CompanyName = Dell, Shell, Facebook`. 47 instantiated slots resulted. For each instantiated slot, we produced a corresponding instantiated text example of how they command may be spoken (as in the last column of the table), yielding 47 *command sentences*. Slot values were selected to exercise known weaknesses of the speech systems: some sounded similar to each other (e.g., `Houston`, `Renton`, `Dell`, `Shell`), some were long (e.g., `The Perks of Being a Wallflower`), some were unconventional (e.g., `Toshi's Teriyaki`) and some came from sets of large cardinality (e.g., we allow all 2680 companies on NASDAQ for `CompanyName` and any number for `Amount` in the currency-conversion intent).

We wrote a Windows Phone application to collect 16bit audio at 16kHz to step subjects through these examples. For each example, we asked subjects to speak up to five variants of how they would request the functionality from a phone. For calibration, they were asked to speak the example sentence provided to them with no change before speaking its five paraphrases. Subjects were encouraged to restrict themselves to language they would use with a phone, provide one sentence per paraphrase, preserve the meaning of the original example

as closely as they could, make no changes to the values of (clearly indicated) slot values when they spoke. To avoid subjects making up exotic paraphrases (a problem we found in an informal test run), we allowed subjects to repeat a previous paraphrase if could not think of a natural paraphrase. They were allowed to delete and re-record any utterance at any time. Subjects were asked to hold the phone 5 to 20 inches from their mouths when speaking. Data was collected in a quiet office with occasional background noise from typing and conversations.

We recruited 20 subjects aged from the twenties to above fifty, 16 male, 4 female who worked at our institution for the study. Subjects were reimbursed $10 each. Data collection was spread over 6 days.

We collected 5653 distinct utterances. Removing empty and malformed files yielded 5401 utterances. We manually transcribed these. Removing utterances that did not follow the above rules removed an left 5038. Further removing the first (calibration) utterance for each sentence left 3998 utterances. Removing all utterances of the calibration sentence (to reduce bias toward the example) left 3781. De-duplication within each person's utterance of each sentence (preserving the first version) yielded 3505 results, roughly equally distributed across the 47 instantiated intents. We use the resulting set (which we call the Spoken Dataset) for our experiments. We believe that it provides a reasonable representation of the diversity of SNL command phrasing, short of an in-situ dataset.

#### 5.1.2 Crowdsourced ("UHRS") Data

An important part of NLify's thesis is that automatically crowdsourced data from the internet can substantially boost the generality of SNL interfaces. Accordingly, we collected a dataset of paraphrases invoking the 27 different intents. In this case, variation in slot values was not important since responses were written and not spoken. We used the Universal Human Relevance System (UHRS), a crowdsourcing service similar to Amazon Mechanical Turk available internally within Microsoft, to collect. UHRS workers were given similar instructions as the spoken data contributors above, except that they were given a scenario description and up to 5 *seed sentences* as examples. These examples were carefully chosen to provide maximal coverage as a developer might. We call these 27x5 = 135 sentences the *Seed Dataset* below.

Workers were asked for up to 3 paraphrases for each scenario at a cost of roughly 3 cents per paraphrase. As with the voice data, filtered the results manually for adhering to the task rules after programatically rejecting ones that mismatched in the number of slots. For the 47 command sentences, we started with 2732 responses of which 2183 had necessary slots, and ended with 1612 unique, well-formed paraphrases that we could use. We call this dataset the UHRS Dataset below.

#### 5.1.3 Machine Paraphrased ("MP") Data

We used the web API of the Microsoft Contextual Thesaurus (MCT) to paraphrases the above 5 hand-selected seed sentences per intent. We requested up to 50 paraphrases per seed sentence and performed automatic cleanup as per Section 4.2.2. The paraphrase engine returned anywhere between 1 and 50 paraphrases across our seeds. Further, its results are usually much closer to the original seed sentence than that of crowd workers. Results are sorted by relevance but scores are not exposed. After a quick manual examination, we picked at most the top 10 results for each seed sentence. We retained 848 non-duplicate paraphrases of 1936 original responses from the engine across the 27 intent areas. We call this dataset the MP Dataset.

### 5.2 Results

#### 5.2.1 Overall Results

To evaluate the SNL recognizers produced by NLify, we assumed a configuration where all 27 intents/commands of Table 1 are installed on the phones of the 20 different users who contributed the Audio dataset. We assume developers of intents contributed the Seed Dataset (roughly 5 templates each), amplified by one or both of the UHRS dataset and the MP dataset. In other words, we trained on Seed + each of {UHRS, MP and UHRS + MP} template sets, tested on the Audio dataset and measured the fraction of intent values and slot values inferred correctly. Although we use a garbage model as per our standard design, all queries are non-garbage in this test. We use the discounted-matching variant of TF-IDF from section 4.3.2.

Figure 8 shows the results across all 27 intents for the best training set configuration (Seed + UHRS). Regarding other configurations (not shown here), whereas the Seed + UHRS configuration shown had a mean intent recognition rate of 85%, adding MP *reduced* the rate to 83.7%, Seed + MP gave only 67% and Seed by itself gave 69%. We discuss machine-paraphrased (MP) amplification in more detail shortly. For each intent, the figure shows the intent and the slot recognition rates side-by-side. Overall, intent recognition rate is a respectable 85% mean ($\sigma = 0.06$). Slot recognition rates are lower at 81% mean ($\sigma = 0.11$). Given that slot recognition is performed deterministically on short acoustic snippets, whereas intent recognition is performed statistically across multiple words, both lower recognition rates and higher variance are unsurprising.

To understand errors better, we created a confusion matrix the intent classifier. 16% of misclassifications were due to confusion between commands 17 and 18 (`ConvertUnits` and `ConvertNumUnits`), another 6% confused `FindTravelTime` and `FindDistance`, and 6% be-

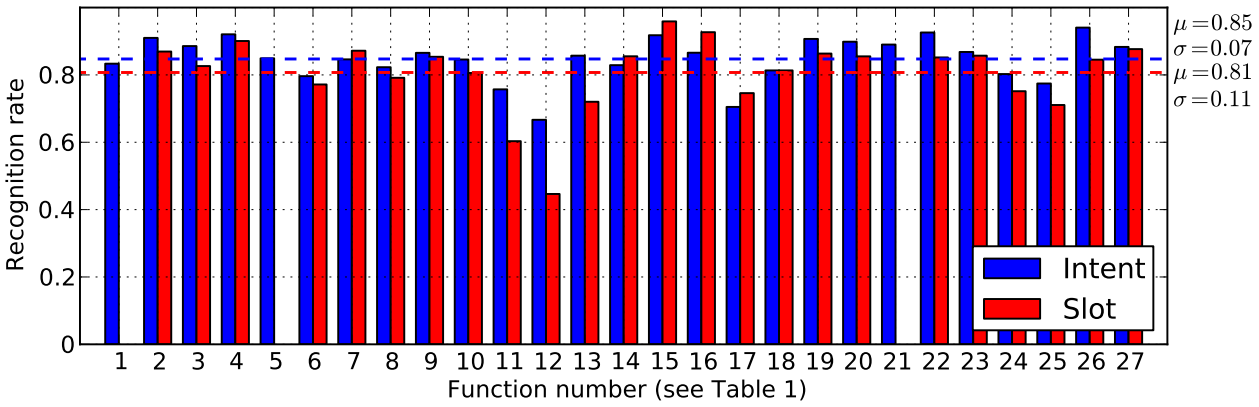



**Figure 8: Overall recognition rates. Dashed lines are averages. Functions numbered per Table 1.**

tween `FindOfficeLocation` and `FindGroup`. These pairs of queries expose the limitations of a word-order-agnostic distance measure such as TF-IDF, since they involve similar words but in different orders. In roughly 12% of other cases, slots were inferred to be of the wrong type: for instance, movie titles were inferred to be currency. Incorrect slot types result in incorrect intents.

To understand errors in slot parsing, we sorted the misparsed slot types by % misparsed. The ones above 10% were `Money` (misparsed 29% of the time), `SrcUnit` (21%), `CompanyName` (20%), `MovieName` (26%), `TgtUnit` (14%), `Amount` (11%). There are two main types of problems. In cases where the result is a phrase, the ability to statistically as opposed to deterministically parse slots would be useful. For instance, "The Perks of Being a Wallflower" may currently be mis-parsed if it misses "Perks", whereas a statistical parse should succeed. In other cases, especially numbers, exact parses are required (e.g., "twenty three" versus "seventy three") and in the usual absence of statistical evidence pointing to either option, the only fallback would be a dialog-based clarification from the user. NLify is currently limited in supporting neither statistical parsing of slots nor dialog.

Finally, machine-paraphrased (MP) data proved to be noisy enough that average recognition rates over all commands was hurt relative to the seed set. For 9 functions (e.g., `SetTimer`), however MP yielded improvements in intent recognition in the 2%–13% range. Given that the MP engine was trained on stock translation corpora (which have little to do with mobile commands), we find this a promising result. Given the promise of instant results, we strongly believe that automated paraphrasing may hold the key to fully automated NLification.

### 5.2.2 *Scaling with number of commands*

Recognition rates for a given command depends on other commands loaded on the phone. How does NLify scale when many such commands compete? To understand the robustness of recognition to such competition in the command set, we picked $n = 10$ random subsets each of $N = 1$, 5, 10, 15, 20 and 27 commands for testing. For each subset, we calculated mean intent and slot recognition rates when trained on the UHRS+Seed data for that subset and tested on the Voice data.

Figure 9 shows the results. The upper line shows intent recognition rates, the lower shows slot recognition rates. Note four points. First, recognition rates do not collapse alarmingly with competition. Second, intent recognition results decline monotonically with amount of competition, which is unsurprising since both the SLM and TFIDF algorithms that identify intents compete across intents. Third, slot recognition does not vary monotonically with number of competitors; in fact the particular competitors seem to make a big difference, leading to high variance for each $N$. On closer examination we determined that even the identity of the competitors does not matter: when certain challenging functions (e.g., 11, 12 and 19) are included, recognition rate for the subset plummets. Larger values of $n$ will likely give a smoother average line. Overall, since slot-recognition is performed deterministically bottom up, it does not compete at the language-model level with other commands.

### 5.2.3 *Impact of NLify Features*

NLify uses two main techniques to generalize from the seeds provided by the developers to the variety of SNL. To capture broad variation, it supports template amplification as per the UHRS and MP datasets. To support small local noise, it advocates a statistical approach (in contrast, e.g., to recent production systems).

We saw in Section 5.2.1 that using the Seed set instead of Seed + UHRS (where Seed has 5 templates per command and UHRS averages 60) lowers recognition from 84% to 69%. Thus UHRS-added templates contribute significantly. To evaluate the incremental value of templates, we measured recognition rates when

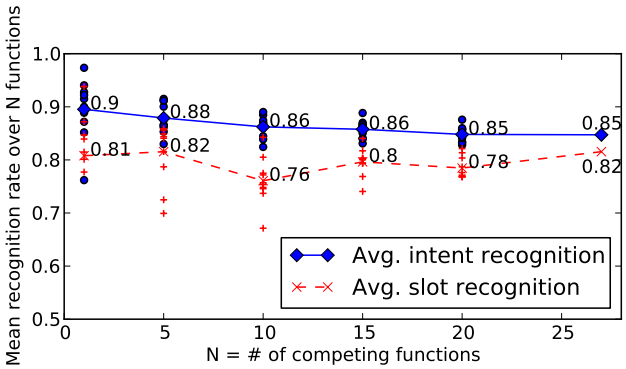



**Figure 9: Scaling with number of commands.**

$f$ = 20, 40, 60 and 80% of all templates were used. We pick the templates arbitrarily for this experiment. The corresponding average recognition rates (across all functions) was 66, 75, 80 and 83%. Figure 10 shows the breakout per function. Three factors stand out: recognition rates improve noticeably between the 80 and 100% configurations, indicating that rates have likely not topped out; improvement is spread across many functions, indicating that more templates are broadly beneficial; and there is a big difference between the 20% and the 80% mark. The last point indicates that even had the developer added an additional dozen seeds, crowdsourcing would still have been quite beneficial.

Given that templates may provide good coverage across paraphrases for a command, it is reasonable to ask whether a deterministic model that incorporates all these paraphrases would perform comparably to a statistical one. Given template amplification, is a statistical model really necessary? In the spirit of the Windows Phone 8 Voice Command system of Section 2, we created a deterministic grammar for each intent. For robustness toward accidentally omitted words, we made the common words {is, me, the, it, please, this, to, you, for, now} optional in every sentence. We compared recognition performance of this deterministic system with the SLM, both trained on the Seed + UHRS data. Figure 11 shows the results for both intent and slot recognition.

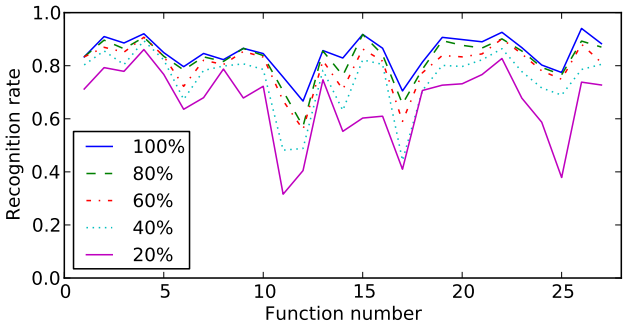
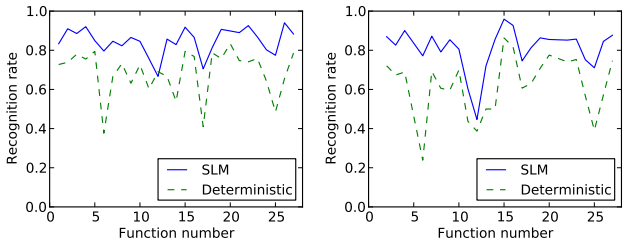



**Figure 10: Incremental benefit from templates.**




(a) intent recognition (b) slots recognition

**Figure 11: Benefit of statistical modeling.**

Two points are significant. First, statistical modeling does add a substantial boost for both intent (16% incremental) and slot recognition (19%). Second, even though slots are parsed deterministically, their recognition rates improves substantially with SLMs. This is because deterministic parsing is all-or-nothing: the most common failure mode by far is that the incoming sentence does not parse, and both slot and intent recognition rates are affected.

As mentioned in Section 4.4, NLify's design decision of using a custom SLM augmented by a mid-sized garbage model instead of a large-vocabulary (LV) language model allows it to run fully on the client. Given that commonly used cloud-based recognizers use much bigger models, it is natural to ask if this independence comes at the cost of recognition accuracy. We therefore use a commercial quality cloud-based voice search model to perform recognition on our Voice dataset before using the TF-IDF engine trained on Seed-UHRS for intent recognition. Since (in the absence of traditional CRF-based recognizer), our infrastructure does not support slot recognition on the resulting text from the LV engine, we replaced the slot values with the slot key names for each LV result. Figure 12 shows the result: the LV model averages 80% intent recognition to NLify's 85%. Customized client-based models a la NLify are therefore at least competitive with *generic* cloud-scale models. These results also suggest that cloud-scale models that allow LM customization per mobile device may receive substantial performance boosts.

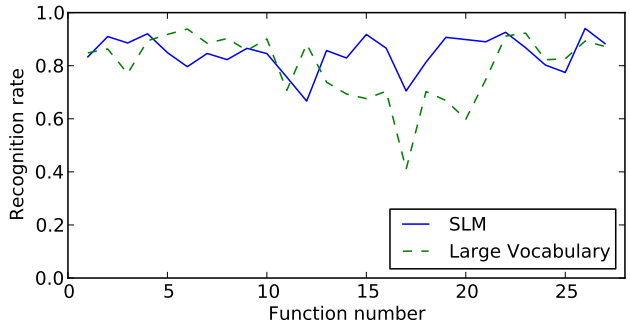



**Figure 12: Comparison to a large LV model.**

The experiments thus far assumed that no query was garbage. In practice, users may speak out-of-grammar commands. NLify's parallel garbage model (PGM) architecture (Figure 6) is set up to catch these cases. Without the PGM, the existing SLM would still reject commands that are egregiously out-of-grammar, or at least assign them low scores. To compare the utility of our PGM design we split our data into two halves. We trained models with and without the PGM on Seed-UHRS data for commands 1 through 13 and tested with Voice queries for all commands (1 through 27). For non-PGM version, if it declared "no recognition" or a score below a carefully selected threshold, we inferred garbage. The score-threshold was selected by eyeballing the data and picking a point that seemed to yield good precision and recall. Our measurements showed that the PGM model has a 83%/8.5% true positive/false positive garbage detection rate whereas the baseline yielded 51%/16%. Garbage models are clearly effective, and could use even more attention.

Finally, our measurements indicate that using our discounted TF-IDF algorithm improved average intent recognition rates by an average of 2.6%, with gains spread over many commands. NLify therefore uses this technique by default.

#### 5.2.4 Resource Usage

Given that the NLify SNL pipeline runs entirely on-client, its resource usage, especially memory and power is important to quantify. We profiled the memory usage of NLify running with 27 intents and the full UHRS + Seed + Slot set of templates on a Nokia Lumia 822 running Windows Phone 8. We determined that the maximum memory usage was 32M, of which roughly 26.5M is consumed by the garbage SLM, 130kB by the SLM and the rest by the slot values. Given that we made no effort to optimize memory layout of the garbage SLM, and current trends in phone memory, we consider this an acceptable result.

| # | Description | Sample commands | stats (rounded) orig. LOC/ addl. LOC/ time taken |
|---|---|---|---|
| 1 | Control a night light | "turn off the light", "higher" | 200 / 50 / 30 min |
| 2 | Get sentiment on Twitter | "review this" | 2000 / 60 / 30 min |
| 3 | Query, control location disclosure | "Where is Alice?","Allow Bob to see me" | 2800 / 60 /40 min |
| 4* | Query weather | "weather tomorrow?" | 3800/ 50 / 70 min* |
| 5* | Query bus service | "When is the next 545 to Seattle" | 8300/ 80 /3 days* |

*Off-the-shelf program, time taken includes understanding program

**Table 2: Developer study programs.**

We further profiled the same phone using a Monsoon Mobile Device Power Monitor as in [12]. The "listening-for-voice" loop on our phone consumed roughly 970mW on average. NLify did not add noticeably to this load: all processing for NLify is effectively piggybacked onto the listening period, which would be required even in competing cloud-based schemes. We conclude that NLify's resource usage on the client is acceptable for daily use.

#### 5.2.5 Developer Study

Given that enabling developers is a key goal of NLify, we conducted a small study among experienced developers to ascertain if NLify is usable and useful. We recruited 5 developers we knew had written Windows Phone programs previously. The developers were all male in the 30-50 age-group, all with over 10 years of programming expertise. Three agreed to modify their existing programs and two opted to use open-source programs. Table 2 gives some details on the programs.

Each developer watched a 3.5-minute video[1] of how to use NLify, used a stock existing program as template, decided what 1 to 3 intents they would add to their application, implemented the functionality (after understanding the program to modify is necessary) and answered 9 questions on a 5-point rating scale about their experience. Responses from UHRS (crowdsourced paraphrases) were delivered to subjects the day after the programming session, since they typically take a few hours to return. Subjects also had the opportunity to provide open-ended comments.

Table 2 gives some details on the programs. Although some of the programs are small, recall that supporting even simple programs that could use speech is an explicit goal of NLify. All programmers completed their task quite quickly with small amounts of additional code. In general, programmers were able to quickly articulate functionality that could benefit from SNL, identify existing functionality in their code that could handle it, and use the NLify IDE to produce result code.

Table 3 details the questions and responses. Overall, developers found the SNL function useful, especially given the value of SNL to their app (question 4). Ease of use could have been better, primarily because our IDE integration was not complete: developers had to add some boilerplate into their code that they do not need in the GUI case. Further, instead of allowing developers to add handlers to the file defining the GUI corresponding to the NLified function, we require a separate parallel file for the NL UI. Both problems are straightforward to address; the broader message is that the SNL UI development process should be as close to the existing GUI process as possible, especially for experienced programmers.

[1] `http://cs.washington.edu/homes/syhan/vids/nlify.wmv`

| # | Question (1-5 scale, 1 most negative) | Score (range/ mean) |
|---|---|---|
| 1 | How useful do you think it is for end-users to have spoken NL interfaces to those features? | 3–4/3.6 |
| 2 | How well did NLifys SNL advertised capabilities match your needs? | 4–5/4.4 |
| 3 | How easy was NLify to use? | 2–4/3 |
| 4 | How easy was NLify to use *relative to the value of the feature(s) you added*? | 3–5/4 |
| 5 | How gracefully did the cost/benefit of NLify scale? Was it clear how to spend more effort to get noticeably better spoken UIs? | 1–3/2.4 |
| 6 | Did you feel crowdsourcing added useful cases to your interface? | 2–4/3 |
| 7 | If you didnt answer Q4 with a 1, did you think the crowd-sourced results were worth spending 3 cents each? (1: $<$ 1 cent, 3: 3 cents, 5: $>$ 10 cents) | 2–4/2.6 |
| 8 | If you didnt answer Q4 with a 1, given your dev cycles, do you think you can afford to wait for crowdsourced results for 3s, 3m, 3h, 12h, 24h | 2–4/2.4 |
| 9 | Did you feel the automatic paraphraser introduced useful variations? | 3–5/3.8 |

**Table 3: Developer study questionnaire.**

The paraphrase engine was surprisingly popular, although our studies showed that it is not consistently useful. On the other hand, perhaps because crowdsourced results were delivered a day later for assessment, they subjects were less enthusiastic about them. A message here maybe that techniques to accelerate crowdsourced results (e.g., [3]) may be important at least in easing adoption of integrated crowdsourcing in development tools.

## 6. CONCLUSIONS AND FUTURE WORK

We have presented, for the first time, a design for a programming system that enables third-party developers to add spoken natural language (SNL) support to their apps. We have presented the first mobile SNL benchmark dataset we are aware of, and used it to drive a comprehensive evaluation of our system. That, and a small developer study, strongly indicate that it is both feasible and useful for third-party developers to add SNL interfaces to their apps.

Much work remains to bring SNL in apps fully into the mainstream, however. For maximum impact, the NLify dispatcher must be integrated into the operating system. Doing so in a secure and efficient way needs study. Our recognition rates for intents and slots are promising, and perhaps adequate for many applications, but much work remains to be done to get error rates closer to 1% than the current 20%. For instance, allowing cloud-based recognizers to be customized per-user-device, using classifiers other than our vector-space model and more careful construction of garbage models are all opportunities. NLify does not provide much support for handling recognition failure. Some form of dialog support to help users recover from a (partially) failed interaction is inevitable, and enabling third-party developers to access this capability is a worthwhile challenge.

## 7. REFERENCES


[1] Microsoft contextual thesaurus. `http://labs.microsofttranslator.com/thesaurus/`. Accessed: 12/7/2012.

[2] C. J. Bannard and C. Callison-Burch. Paraphrasing with bilingual parallel corpora. In *ACL*, 2005.

[3] M. S. Bernstein, J. Brandt, R. C. Miller, and D. R. Karger. Crowds in two seconds: enabling realtime crowd-powered interfaces. In *UIST*, pages 33–42, 2011.

[4] F. A. Bishop. Speech-enabling a windows phone 8 app with voice commands. *MSDN Magazine*, 27(11), November 2012.

[5] D. L. Chen and W. B. Dolan. Collecting highly parallel data for paraphrase evaluation. In *ACL*, pages 190–200, 2011.

[6] W. B. Dolan and C. Brockett. Automatically constructing a corpus of sentential paraphrases. In *Third Intl. Workshop on Paraphrasing*, 2005.

[7] S. T. Dumais et al. Inductive learning algorithms and representations for text categorization. In *CIKM*, 1998.

[8] L. Fei-Fei, A. Iyer, C. Koch, and P. Perona. What do we perceive in a glance of a real-world scene? *J. Vision*, 7(1):1–29, 1 2007.

[9] S. Hahn et al. Comparing stochastic approaches to spoken language understanding in multiple languages. *IEEE Trans. Audio, Speech & Language Processing*, 19(6):1569–1583, 2011.

[10] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Pearson Prentice Hall, second edition, 2008.

[11] S. Kok and C. Brockett. Hitting the right paraphrases in good time. In *NAACL-HTL*, pages 145–153, 2010.

[12] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *MOBICOM*, pages 317–328, 2012.

[13] T. Paek, S. Gandhe, D. Chickering, and Y. Ju. Handling out-of-grammar commands in mobile speech interaction using backoff filler models. In *SPEECHGRAM*, 2007.

[14] R. Rosenfeld. Two decades of statistical language modeling: Where do we go from here? In *Proc. of the IEEE*, 2000.

[15] G. Salton et al. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, Nov. 1975.

[16] G. Saon and J.-T. Chien. Large-vocabulary continuous speech recognition systems: A look at some recent advances. *Signal Processing Magazine, IEEE*, 29(6):18 –33, nov. 2012.

[17] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[18] W. Wang, D. Bohus, E. Kamar, and E. Horvitz. Crowdsourcing the acquisition of natural language corpora: Methods and observations. In *SLT*, 2012.

[19] J. Williams. A case study of applying decision theory in the real world: Pomdps and spoken dialog systems. In L. Sucar, E. Morales, and J. Hoey, editors, *Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions*. IGI Global, 2011.