NLify: Lightweight Spoken Natural Language Interfaces via Exhaustive Paraphrasing

Seungyeop Han
University of Washington

Matthai Philipose
Microsoft Research

Yun-Cheng Ju
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 standalone mobile applications. The central challenge is to create statistical recognition models that are accurate and resource-efficient in the face of the variety of natural language, while requiring little specialized knowledge from developers. We show that given a few examples from the developer, it is possible to elicit comprehensive sets of paraphrases of the examples using internet crowds. The exhaustive nature of these paraphrases allows us to use relatively simple, *automatically derived* statistical models for speech and language understanding that perform well without per-application tuning. 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.

Author Keywords

Spoken natural language interface; third-party mobile applications; crowdsourcing; paraphrasing.

ACM Classification Keywords

H.5.2. Information Interfaces & Presentation (e.g. HCI): User Interfaces—Natural language; D.2.6. Software Engg. : Programming Environments—Integrated environments

INTRODUCTION

Visual attention and screen real estate are at a premium in mobile devices. Spoken natural language (SNL) interfaces, which allow users to express their functional intent verbally without conforming to a rigid syntax, have therefore been of increasing recent interest. Prominent "first-party" systems such as Siri and Google Voice Search offer such functionality on select domains today, making extensive use of cloud-based servers. In this paper, we present a system that enables any ("third-party") developer to build SNL interfaces that run on resource-light sporadically connected devices. Although we focus on a mobile phone based implementation, we believe that this combination of ease of development and low resource usage will be of equal appeal and relevance to speech-enabling other ubiquitous devices.

We target command-and-control (C&C) style interactions (e.g., "When is the next 210 bus to San Jose?") supported by today's first-party systems. 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 *paraphrases*, e.g., "When's the 210 to San Jose expected?", or even "Next San Jose 210 bus". The role of the SNL system is to analyze the speech and dispatch it to the correct handler, with parameters (also known as *slots*, e.g., "210" and "San Jose") correctly instantiated. Although single-intent C&C is simple relative to the full range of spoken natural interactions such as dialog and dictation, it is both broadly useful and considered impractical for non-experts to implement.

Third-party SNL interfaces pose several challenges. First, the task of developing these interfaces, traditionally the domain of speech and natural language specialists, must be simplified enough to be accessible to lay developers. Second, when these interfaces are in use, language-processing computation must ideally be completed wholly on the phone. The additional recurring cost implicit in using cloud-based servers would pose a significant barrier-to-use for third parties. Further, as spoken interaction becomes more common, spotty high-speed connectivity [1] becomes an ever larger limiter.[1] Third, given that apps are developed independently of each other and only installed by end users, the interface must allow apps to specify the underlying speech and language classifiers in a modular way while still allowing fast install by users. Fourth, the system must provide an appropriate mental model for a user talking to these apps. Currently-popular models such as valets or search assistants that allow users to "say anything" to the system imply a (cloud-based) large-vocabulary speech decoder while at the same time obfuscating the specific apps installed on the phone.

We present NLify, a fully implemented system that addresses the above challenges using a combination of carefully considered algorithmic and system design choices. The fundamental difficulty in developing natural language interfaces is that of paraphrasing: a single command may be phrased in hundreds of alternate ways. Once an app is widely adopted, data from its users can be a source of new paraphrases. However, NLify seeks to address the important problem of providing

---

[1]Since many apps use web-based services to handle requests, spotty connectivity may seem to render them useless in any case. However, in many of these cases (e.g., maps, calendars, weather, movies, transit, music, unit conversions, phrase books) small *sporadically updated* databases cached on the phone suffice. Such weakly connected operation is important and, we believe, will be increasingly prevalent given user expectations.
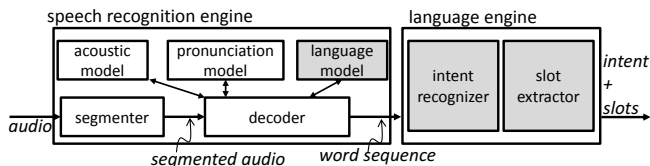
Figure 1. Standard SNL pipeline. NLify focuses on the gray boxes.

acceptable SNL *out of the box*. The main idea behind NLify is *exhaustive paraphrasing*. The intuition is that we automatically generate paraphrase sets covering "all" ways of saying the command so that speech models can be derived solely from them and intent-classification algorithms need to perform very modest generalization. In particular, we show how to design a crowdsourcing system to obtain large paraphrase sets for each command. We further show how to embody this functionality in an integrated development environment, so that the workflow for a programmer is similar to developing graphical interfaces, with no knowledge of crowds required.

We advocate a simple *natural language shortcut* metaphor for what users can expect to say to their phones: essentially, for each application they have installed, they can expect to access its key functionality by voice. We do not support other queries (e.g., "Siri, what color are your eyes?"). The restriction to shortcuts allows NLify to make a closed-world assumption: all utterances are either targeted at commands with local paraphrase sets or are out of vocabulary. The assumption allows all commands to be processed locally on the phone by a combination of compact statistical models derived from the paraphrases. Further, exhaustive paraphrasing facilitates an architecture that allows new applications to represent their incremental functionality modularly as paraphrase-set/handler pairs, while maintaining fast speech and language model update times at app install time.

NLify is the first (fully realized) proposal for a system to support third-party development of spoken natural language interfaces for mobile devices. The exhaustive-paraphrasing approach implemented via automated crowdsourcing is novel. The derived architecture allowing modular addition of applications and purely local recognition is new. We are unaware of a dataset to test spoken natural language C&C. We present a new and extensive benchmark dataset including roughly 3500 spoken utterances from 20 subjects targeted at 27 different intents, with 1600 crowdsourced paraphrases for these intents. A quantitative evaluation of our implementation shows overall recognition rates of 85/81% across intents/slots, along with drill-down analysis of the impact of our design decisions on accuracy. We further show that our specialized on-phone SNL interface is competitive in accuracy with a generic cloud-based large-vocabulary system. Also, it consumes little power beyond that already spent by the phone's audio collection loop. Finally, a small qualitative programmer study gauges NLify's usability and utility.

BACKGROUND
We describe the typical spoken natural language (SNL) pipeline and its relation to NLify. Figure 1 illustrates a standard pipeline for inferring intent from audio. Audio signals (sampled at 16-48kHz and 8-16 bits) are segmented into sentences

that represent commands. Segments are converted into words and then intents/slots as described below.

Speech Recognition Engine
Given streaming audio, the speech recognition (SR) engine first segments it using heuristics; NLify assumes speak-on-button-press followed by terminate-speech-on-silence.

The engine is parameterized by three statistical models. The *acoustic model* captures the joint distribution over feature vectors derived from raw audio and *phones*, which can be thought of as the primitive units of the sounds in a language. The *pronunciation model* captures the mapping between phones and words. Neither model usually changes with applications, and NLify leaves them untouched.

The *language model (LM)* represents sequential correlation among words expected to be input to the SR engine. Of the three models parameterizing the recognition engine, the LM most affects per-application SNL recognition performance. For example, the distribution of phrases to control an alarm application is very different from that for making a phone call. Ideally, the correlations are derived from training data representing expected input. When data is sparse (e.g., only a few paraphrases are available as in [5]) correlations are represented as deterministic grammars that are essentially the union of the paraphrases. In the data-rich setting, the models could be statistical distributions over subsequences ("n-grams") of words, termed Statistical Language Models (SLMs). Given enough training data, SLMs are known to be more robust to variation in phrasing and noise than deterministic models, and are preferred for SNL modeling [22].

Training data, however, is usually lacking in two ways. First, slots could have very many values (e.g., numbers, dates or movie names), so that most slot values will be missing from training data. Such data is usually provided as separate domain specific grammars. The domain-specific data is accommodated by incorporating a hierarchical component to the statistical model that allows these grammars to be parsed as part of recognition [25]. NLify uses such a hierarchical model in an important way explained in the next section.

More fundamentally, different phrasings of the command may be missing from training data (e.g., a test phrase may say "go ahead and give me the date", whereas training data for the command may not contain "go ahead"). In this case, the state-of-the-art approach is to assume that domain-specific training data is significantly lacking in variety, and to mix the domain-specific training data with a general ("large-vocabulary") statistical model [14] for the language (e.g., English) in which commands are spoken. Good large-vocabulary recognition systems today have resource requirements that are designed for the cloud rather than mobile devices [7], thus precluding the standalone recognition we seek. NLify's exhaustive paraphrasing approach instead attacks the assumption that training data must be spotty, and uses solely this rich, in-domain, training set to generate its speech model. It uses the newly available resource of crowd workers in a convenient and efficient way to do so. In principle, this model could be complemented by a compact general model on the phone.
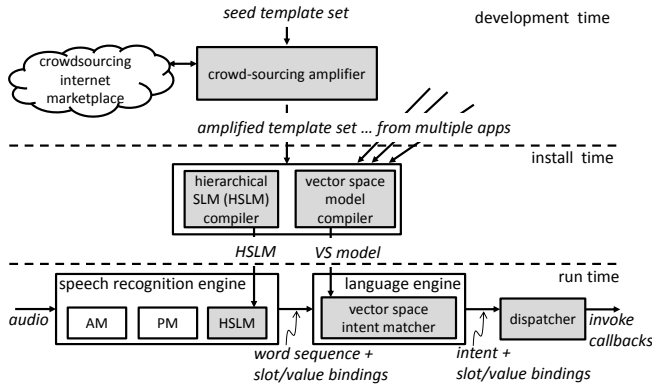
Figure 2. NLify system architecture.

Language Engine

Functionally, the language engine is a classifier from text sequences to intents and slots. The consensus [11, 21] is that the conditional random field (CRF) is the classifier of choice to implement the language engine. Typically, the CRF designates each word (or phrase) of the incoming sentence as slots or carrier phrases (i.e., the "filler"), and infers the intent of the sentence as a whole. CRFs are especially highly rated because their discriminative nature allows inferred values to be conditioned on arbitrary combinations of incoming words (known as "features"), unlike their generative counterparts [21]. In fact adding the right features to compensate for adequate training data is one of the key roles of a specialist in tuning models for natural language understanding. It is common for CRFs for SNL to be trained over hundreds of thousands of features [11] (these are usually various assignments for the size-n subsequences, or "n-grams" surrounding the current word). Training these models often takes tens of seconds on a desktop-class processor even with the fastest current implementations of CRFs [18].

NLify takes the point of view that its users are not sophisticated enough to add new features to the classifier. The flexibility of CRFs is therefore much less useful. On the other hand, we expect exhaustive paraphrasing to provide enough data that even simple (e.g., nearest-neighbor style) classifiers will do an acceptable job of classification. Further, since the system-wide classifier must be re-trained every time a user installs a new application, it is critical that training be very fast, e.g., under 5 seconds. Once again, by enabling simple non-parametric classifiers that are fast to compute but rely on adequate data, exhaustive paraphrasing comes to the rescue. By relying on "big data" therefore, NLify trades off expert-friendly and relatively slow-to-learn classifiers for a developer-friendly, fast-to-learn variant.

RELATED WORK

Several ubicomp systems have targeted mobile audio. The Personal Audio Loop [13] allowed users to save audio of conversation snippets of interest. Features from spoken audio have been shown to reveal social dynamics [27], stress levels [16] and certain human activities [28]. Predicting silent periods in speech has been shown to allow radios to be shut down, extending phone battery life [20]. None of these systems recognizes speech or processes natural language, much

less enables non-expert developers to develop systems that do so efficiently in real time, the core contribution of NLify.

Many efforts have examined spoken natural language in the context of interacting with devices in smart spaces. Transcribing text from speech followed by simple sentiment analysis is shown to have promise in eliciting community feedback [26]. The speech recognition in this case is human aided (taking "minutes" to "hours"), although language analysis is automated if rudimentary. Other formative work has shown that a home computer, if speech enabled, may be quite useful [6]; however, the researchers noted that even the basic keyword-based interface for their research system was hard to build. Finally, failure-mode analysis and dialog-based recovery for spoken interfaces clearly show great promise [19]. However, this research too features a human-powered (Wizard of Oz) speech system. In summary, much formative work has pointed out the potential value and difficulty of spoken natural language. NLify provides technical machinery to help realize this potential.

SYSTEM DESIGN & IMPLEMENTATION

Figure 2 shows how NLify is structured. NLify acts in three phases. When apps are being developed (*development time* in the figure), given a small set of textual seed examples of utterances ("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. 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 the inferred intent.

Template definition language

NLify provides developers a two-level grammar definition language to define *template pattern sets* (TPSs):

$T \in \text{TPS} \leftarrow h(s_1, \ldots, s_m) \quad d \quad t_1, \ldots, t_n \quad g_1, \ldots, g_k$
$h \in$ 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 slot variables. These variables must be the root of a deterministic context-free grammar g. Slot variables 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"
```

Learning models from templates

Given a set T of templates, NLify learns a hierarchical SLM and a vector-space model for intent classification from them. Note that all relevant templates are only available at install time, so that learning must be fast.

Learning the SLM

NLify uses a conventional trigram-based representation of sequential correlation. A trigram model is the conditional 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 *smoothing* techniques for interpolating these probabilities from bigram and unigram probabilities have been proposed [15]. For speed, we use a simple linear smoothing scheme. Learning the smoothed trigram is dominated by the cost of counting the co-occurrence of various triples, pairs and words in T and is extremely fast (below a second even for several ten thousand templates). The resulting models fit within 2MB.

To handle out-of-vocabulary commands, NLify runs a "garbage" SLM (about 30MB big) generated from a medium-sized corpus independent of the spoken-command domain, in parallel with the above SLM. When a command is decoded fully, the higher scoring of the garbage model and foreground SLM is selected as the output of the compound SLM. Although more sophisticated techniques exist for handling out-of-vocabulary speech [3], NLify chooses this simple approach under the assumption that mobile commands are distinctive relative to other speech. Further, it assumes that its template set is exhaustive enough that the garbage model will not dominate the primary SLM incorrectly.

Note that NLify templates are two-level, with slots s in templates defined by a separate grammar g. This two-level structure is used to produce a hierarchical SLM. During statistical matching of the higher-level trigram model, slots s in templates trigger *deterministic processing* using the associated grammar g, so that slots are discovered during speech recognition. NLify thus moves slot extraction, traditionally a role of the language engine, into the speech engine. In fact, the local n-gram processing by the SLM engine to infer the identity of slots is not dissimilar from that performed by the n-ary potential functions of CRFs, although the SLM version is fundamentally less powerful because it is one-pass. By conflating the second level of the SLM with slot inference, NLify avoids needing a separate CRF for finding slots, at the potential cost of higher classification error.

Learning the Intent Classifier

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) in T that it "best matches". The target intent/handler is simply the one associated with the best matching template. We use a standard vector-space model for document [23] with Term-Frequency Inverse-Document-Frequency (TF-IDF) weighting. Relevant details follow. 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 per token 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 λ_{ij} of token τ_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 λ rewards words that appear often in the document being featurized (high f) and penalizes appearance in many documents.

This (standard) formulation of TF-IDF is 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 $T_j = \{t|t \in \text{TPS}_j\}$.

Template Set Amplification

Programmers typically provide 3-5 examples in the seed template set. NLify enlarges this set automatically 10-100x toward exhaustive paraphrasing. Automatic machine paraphrasing is an old, hard problem in natural language processing (NLP) [2]. We attack it by finessing the problem by using people instead of machines to paraphrase.

Crowdsourcing services such as Amazon Mechanical Turk have been used to collect paraphrase sets that serve as NLP benchmarks [8, 9, 24]. Essentially, workers worldwide are paid tiny amounts to paraphrase individual example sentences or concepts. NLify shields its developers from the intricacies of programming crowd services by automatically deriving and issuing paraphrasing tasks and post-processing them to yield good results.

Developers need only register an account at the paraphrasing site and provide NLify the key. NLify submits the scenario description d from the TPS with a subset of templates from T to the crowd 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].
```
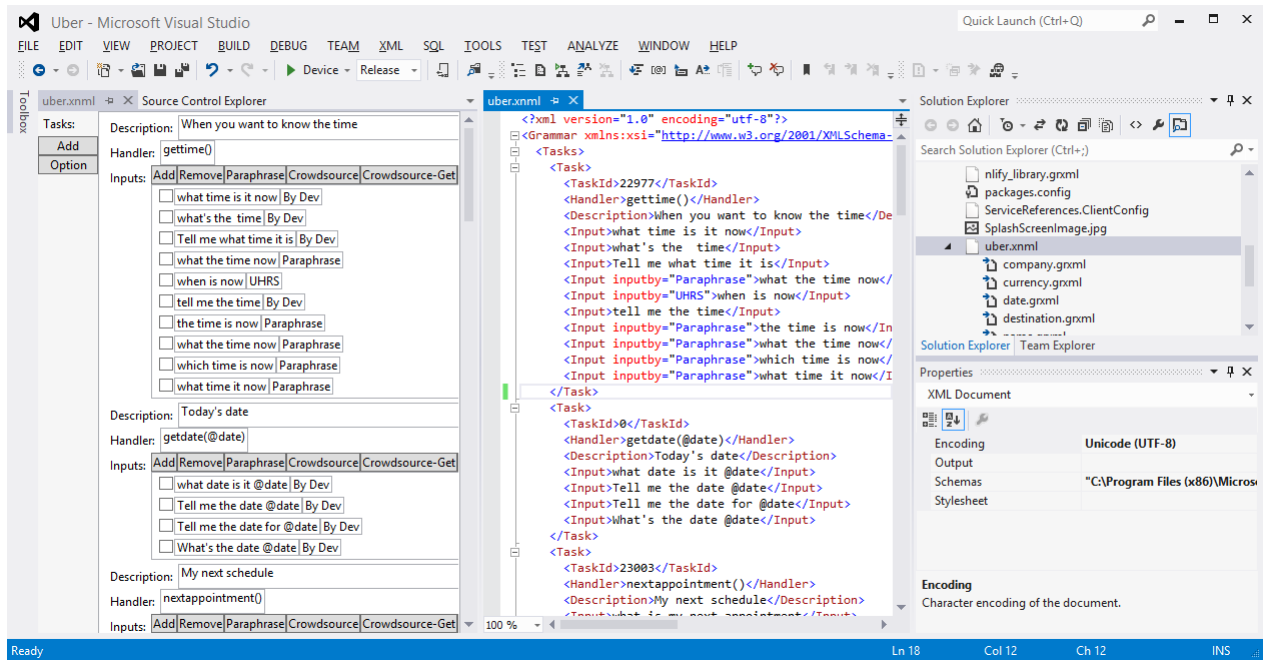
Figure 3. IDE extension for NLify.

```
Example 2:  How much was [Facebook] stock [yesterday]?
Enter five other ways you may speak in this scenario.
```

NLify provides a generic paraphrasing-task description to each crowd worker. The description specifies 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 and to use all slot values in the input sentence. NLify presents results back 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.

Static automated crowdsourcing substantially reduces the effort of using the crowd for paraphrasing. However, even early NLP benchmarking efforts recognized that the way a task is presented biases the paraphrases returned by the crowd [9]: e.g., if the example says "what is my next meeting", responses that substitute "appointment" for "meeting" are relatively uncommon. We added the "scenario" field to our tasks to reduce bias toward examples as per [24]. It seems manageable for developers to provide scenarios. On the other hand, we rejected other expert-recommended techniques to reduce bias such as using images [10] and videos [8] of the task instead as being too onerous for the developer.

Implementation
NLify is implemented as an extension to Microsoft Visual Studio 2012 complemented by the publicly available Microsoft Speech Recognition API for Windows Phone 8 (SAPI) as its speech engine. Figure 3 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 `Crowdsource` button 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. A dozen NLify-based apps have been working on Windows Phone for over six months.

EVALUATION
We seek to answer the following questions. 1. How well 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 do on-phone implementations compare to cloud-based ones? 5. What is the resource consumption of NLify? 6. What is the developer experience using NLify?

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 across 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

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.

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 slots (e.g., Amount) were numbers, time durations, or currencies of unbounded cardinality implemented as external rules. We then collected three datasets.

Voice ("Audio") Dataset
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 the 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., 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 16-bit 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 orig-

inal 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 they 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.

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 5401 non-empty, non-malformed audio files. We manually transcribed these and mechanically cleaned as follows. Removing utterances that did not follow the above rules left 5038. Further removing all utterances of the calibration sentence yielded 3505 results, roughly equally distributed across the 47 instantiated intents. We use the resulting set (which we call the *Audio 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.

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 the dataset. UHRS workers
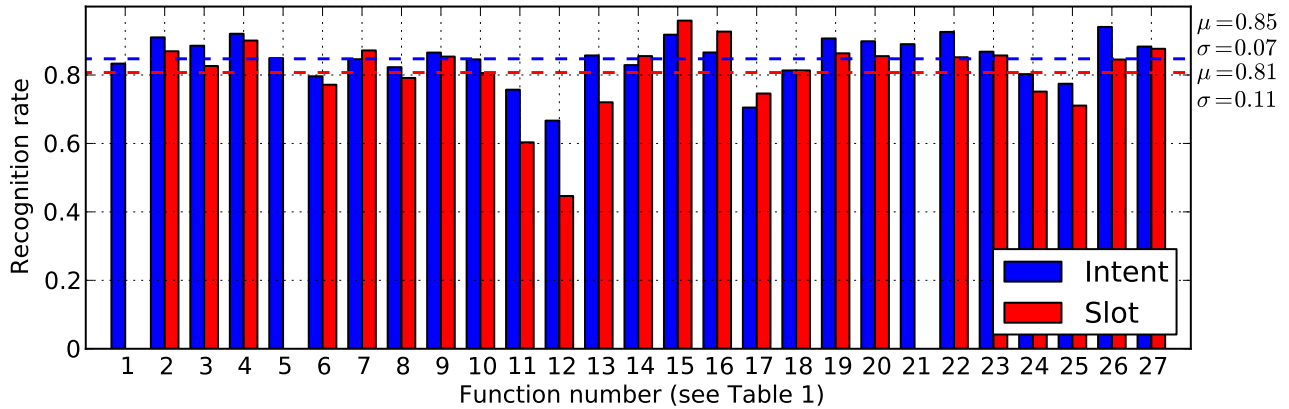
Figure 4. Overall recognition rates. Dashed lines are averages. Functions numbered per Table 1.

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. The scenario description was specifically crafted to be distinct from the descriptions and examples used to elicit the Audio Dataset. The seed sentences were created, solely using this scenario description, by a researcher isolated from the speech data collection effort. 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, the above researcher mechanically filtered out non-adherence to rules and duplicates. For the 47 command sentences, we started with 2732 responses and ended up with 1612 unique, well-formed paraphrases that we could use. We call this dataset, corresponding to our static automated crowdsourcing technique, the *UHRS Dataset* below.

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 the UHRS dataset. In other words, we trained on the Seed + UHRS template set, 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 unless specified. We use the discounted-matching variant of TF-IDF unless specified.

Figure 4 shows the results across all 27 intents for the best training set configuration (Seed + UHRS). Whereas the Seed + UHRS configuration shown had a mean intent recognition rate of 85%, Seed by itself gave 69%. 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, lower recognition rates and higher variance are unsurprising.

To understand errors better, we generated a confusion matrix for 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% between `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 currently does not support either statistical parsing of slots or dialog.

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 and tested on the Voice data.

Figure 5 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 unsur-
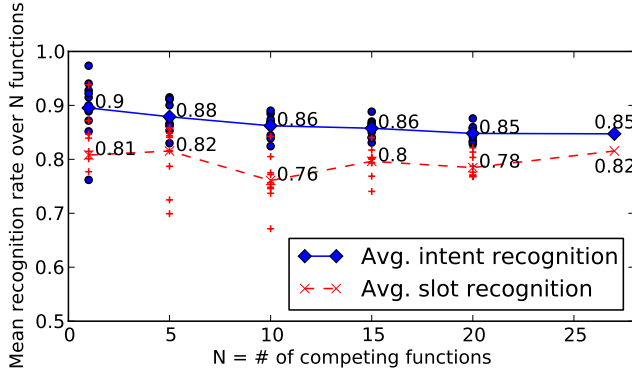
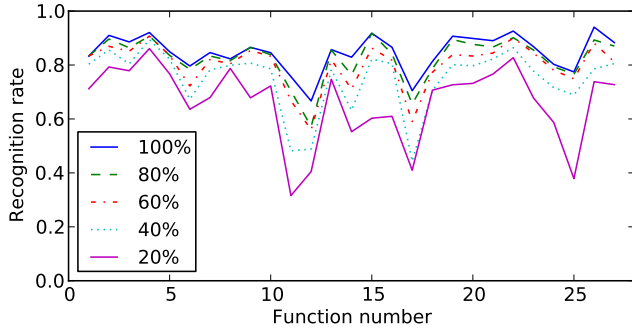**Figure 5. Scaling with number of commands.**

Figure 6. Incremental benefit from templates.

(a) intent recognition   (b) slots recognition

Figure 7. Benefit of statistical modeling.

Figure 8. Comparison to a large vocabulary model.

prising since both the SLM and TF-IDF 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.

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 dataset. To support small local noise (e.g. words dropped in the speech engine), it advocates a statistical approach even when the models are run locally on the phone (in contrast, e.g., to recent production systems [5]).

We saw earlier that using the Seed set instead of Seed + UHRS (where Seed has 5 templates per command and UHRS averages 60) lowers recognition from 85% to 69%. Thus UHRS-added templates contribute significantly. To evaluate the incremental value of templates, we measured recognition rates when 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 6 shows the breakout per function. Three factors stand out: recognition rates im-
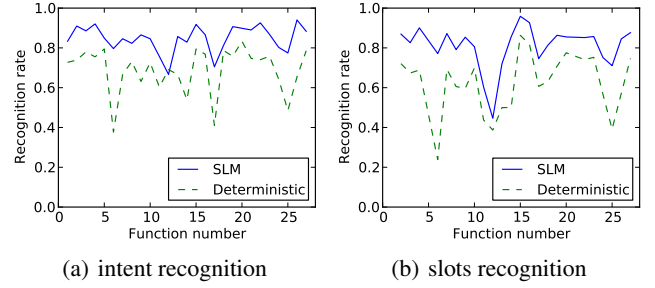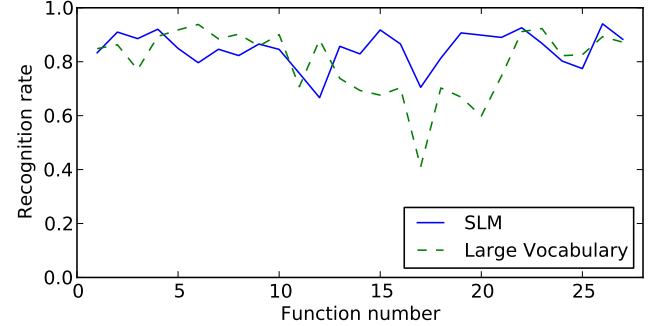
prove noticeably between the 80 and 100% configurations, indicating that rates have likely not topped out; improvement is spread across many functions, indicating that more templates are broadly beneficial; and there is a big difference between the 20% and the 80% mark. The last point indicates that even had the developer added an additional dozen seeds, crowdsourcing would still have been 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 [5], 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 7 shows the results for both intent and slot recognition. 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, affecting both slot and intent recognition rates.

The experiments thus far assumed that no query was garbage. In practice, users may speak out-of-grammar commands. SLify's parallel garbage model architecture is set up to catch these cases. Without the garbage model, the existing SLM would still reject commands that are egregiously out-of-gra-

#	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 didn't 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 didn't 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

Table 2. Developer study questionnaire.

mmar, or at least assign them low scores. To compare the utility of the garbage model we split our data into two halves. We trained models with and without the garbage model on Seed-UHRS data for commands 1 through 13 and tested with Voice queries for all commands (1 through 27). For the no-garbage-model 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 garbage model has a 83%/8.5% true positive/false positive garbage detection rate whereas the baseline yielded 51%/16%. Garbage models are clearly effective.

Comparison to Cloud Implementation

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 8 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 considering cloud-scale models that allow SLM customization per mobile user.

Resource Usage

Since 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. The maximum memory usage was 32M, of which roughly 26.5M is the garbage SLM, 130kB the Nlify SLM and the rest slot values. Given our minimal effort to optimize memory layout, we consider this acceptable.

We further profiled the same phone using a Monsoon Mobile Device Power Monitor as in [17]. The "listening-for-voice" loop on our phone consumed roughly 970mW on av-

#	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 3. Developer study programs.

erage. 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.

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 3 details the programs.

Each developer watched a 3.5-minute video 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 if necessary) and answered 8 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 3 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 code.

Table 2 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 as possible to the existing GUI process.

Although developers found the results of crowdsourcing to be useful, the delay of several hours between requesting crowdsourcing results and receiving them was consistently reported as needing improvement. A message here maybe that techniques to accelerate crowdsourced results (e.g., [4]) may be important at least in easing adoption of integrated crowdsourcing in development tools.

CONCLUSIONS AND FUTURE WORK
We have presented the first design for, and a complete implementation of, a programming system that enables third-party developers to add spoken natural language (SNL) support to their apps. The system runs efficiently on mobile devices. 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, indicate that it is both feasible and useful for third-party developers to add SNL interfaces to their apps.

Many obvious directions remain for improving and validating NLify, however. Given NLify's promise on lab-collected end-user data, a deployment study gauging its effectiveness in daily use is a clear next step. Anecdotally, NLify has been demonstrated in several conference-demo-style settings [12], and has functioned quite well despite loud background noise, given phones held close to the mouth or a wearable speech accessory. Beyond testing, refining how users start and stop speech, interact with wearable accessories, control privacy and correct errors, is critical. Orthogonally, system performance can still stand to improve. Crowdsourcing needs to become more efficient and lower latency. Improvements in classification should yield recognition rates well above the current 85%, with support for correction dialog when the inevitable errors occur. All that said, we believe that NLify as it is today is sufficient for prototyping useful and useable mobile SNL interfaces.

ACKNOWLEDGMENTS

REFERENCES
1. Balasubramanian, A., et al. Augmenting mobile 3g using wifi. In *Mobisys* (2010).
2. Barzilay, R., and McKeown, K. Extracting paraphrases from a parallel corpus. In *ACL* (2001).
3. Bazzi, I. *Modeling Out-of-Vocabulary Words for Robust Speech Recognition*. PhD thesis, MIT, 2002.
4. Bernstein, M. S., et al. Crowds in two seconds: enabling realtime crowd-powered interfaces. In *UIST* (2011), 33–42.
5. Bishop, F. A. Speech-enabling a windows phone 8 app with voice commands. *MSDN Magazine 27*, 11 (November 2012).
6. Brush, A. J., Johns, P., Inkpen, K., and Meyers, B. Speech@home: an exploratory study. In *CHI EA* (2011).
7. Chelba, C., et al. Large scale language modeling in automatic speech recognition. Tech. rep., Google, 2012.
8. Chen, D. L., and Dolan, W. B. Collecting highly parallel data for paraphrase evaluation. In *ACL* (2011), 190–200.
9. Dolan, W. B., and Brockett, C. Automatically constructing a corpus of sentential paraphrases. In *3rd Intl. Work. on Paraphrasing* (2005).
10. Fei-Fei, L., et al. What do we perceive in a glance of a real-world scene? *J. Vision 7*, 1 (1 2007), 1–29.
11. Hahn, S., et al. Comparing stochastic approaches to spoken language understanding in multiple languages. *IEEE Trans. Audio, Speech & Language Processing 19*, 6 (2011), 1569–1583.
12. Han, S., et al. Nlify: Mobile spoken natural language interfaces for everyone. In *HotMobile (demo)* (2013).
13. Hayes, G. R., et al. The personal audio loop: Designing a ubiquitous audio-based memory aid. In *Mobile HCI* (2004).
14. Hsu, B.-J. P. *Language Modeling for Limited-Data Domains*. PhD thesis, MIT, 2009.
15. Jurafsky, D., and Martin, J. H. *Speech and Language Processing*, second ed. Pearson Prentice Hall, 2008.
16. Lu, H., et al. Stresssense: detecting stress in unconstrained acoustic environments using smartphones. In *UbiComp* (2012).
17. Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *MOBICOM* (2012), 317–328.
18. Okazaki, N. Crfsuite: a fast implementation of conditional random fields (crfs), 2007.
19. Oulasvirta, A., et al. Communication failures in the speech-based control of smart home systems. In *Intelligent Environments* (2007), 135–143.
20. Pyles, A. J., et al. Sifi: exploiting voip silence for wifi energy savings insmart phones. In *UbiComp* (2011).
21. Raymond, C., and Riccardi, G. Generative and discriminative algorithms for spoken language understanding. In *INTERSPEECH* (2007).
22. Rosenfeld, R. Two decades of statistical language modeling: Where do we go from here? In *Proc. of the IEEE* (2000).
23. Salton, G., et al. A vector space model for automatic indexing. *Commun. ACM 18*, 11 (Nov. 1975), 613–620.
24. Wang, W., et al. Crowdsourcing the acquisition of natural language corpora: Methods and observations. In *SLT* (2012).
25. Wang, Y.-Y., et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE 29*, 6 (nov. 2012), 82 –97.
26. Whittle, J., et al. Voiceyourview: collecting real-time feedback on the design of public spaces. In *Ubicomp* (2010).
27. Wyatt, D., et al. Towards the automated social analysis of situated speech data. In *UbiComp* (2008).
28. Yatani, K., and Truong, K. N. Bodyscope: a wearable acoustic sensor for activity recognition. In *UbiComp* (2012).