Idiolect: A Reconfigurable Voice Coding Assistant

Breandan Considine
McGill University
bre@ndan.co

Nicholas Albion
Independent Developer
nalbion@yahoo.com

Xujie Si
University of Toronto
xsi@cs.utoronto.ca

***Abstract*—This paper presents Idiolect, an IDE plugin for voice coding and a novel approach to building bots that allows for users to define custom commands on-the-fly. Unlike traditional chatbots, Idiolect does not pretend to be an omnicient virtual assistant but rather a reconfigurable voice programming system that empowers users to create their own commands and actions dynamically, without rebuilding or restarting the application. We present a case study of integrating Idiolect [1] with the IntelliJ Platform, illustrate some example use cases, and offer some lessons learned during the tool's development.**

***Index Terms*—speech recognition, voice programming, bots**

I. Introduction

Humans are able to quickly learn new words and phrases, and apply them in a variety of contexts. Chatbots, on the other hand, are often limited to a set of static commands and phrases defined at compile-time. This can be frustrating for users, who struggle to express their intent, as well as bot-developers, who must anticipate user intent and write bindings for each new capability. This rigidity is a common source of misalignment between user intent, author expectations and bot capabilities.

Voice coding allows users to quickly dictate their own commands and phrases without resorting to a Turing Complete programming language. For example, the user might say "whenever I say *open sesame*, do the following action", thereafter, the system will perform the desired action when so instructed. Or, "whenever I say *redo thrice*, repeat the last action three times". Or, e.g., invoke a function in a scripting language, open a file, or perform other tedious chores.

Idiolect provides a default lexicon of phrases, but does not force users to learn them explicitly. Instead, we allow users to override the default settings with their own voice commands on-the-fly, which are incorporated into the lexicon immediately. This shifts the burden of adaptation to the system, freeing users to express their intent in a natural way.

Primarily, Idiolect observes the following design principles: be (1) natural to use, (2) easy to configure, (3) as unobtrusive as possible. We believe that these principles are important for a system intended to be used by developers, who are busy people and capable of configuring the system themselves. We also support developers with visual and motor impairments, who may have difficulty typing, or prefer to use a voice interface.

In the following paper, we describe Idiolect, a dynamically reconfigurable voice-programming system that supports teaching the IDE new commands and actions on-the-fly, by either verbally or programmatically expressing the desired behavior.

II. Prior Work

Mary Shaw, during her 2022 SPLASH keynote called for programming languages to address the needs of "vernacular developers". Jin Guo has also talked about the need for programming in "ordinary people's langauge". We take their proposals quite literally to mean that computers should be able to interpret spoken programs, and not just written ones.

Early attempts to build voice programming systems can be traced back at least twenty years to Leopold and Amber's (1997) work on keyboardless programming, later revisited by Arnold and Goldthwaite (2001), Begel and Graham's (2005) and others. These systems allow users to write code by speaking into a microphone, however early voice programming systems were limited by a small vocabulary, and do not consider IDE integration or reconfigurability.

Another stream of work has explored teaching voice assistants to use custom phrases (Chkroun & Azaria, 2019). Their approach is similar to our own, but is limited to a single user, and does not consider more general forms of voice programming. It also predates most of the recent progress on large language modeling, which we consider a transformative enabling technology for speech programming.

III. Speech Models

Today, automatic speech recognition (ASR), the translation of an audio waveform containing speech to text, is essentially a solved problem - one of the many pretrained ASR models would work well enough for our purposes. However, we also require realtime offline speech recognition capabilities built on an open source deep speech pipeline, which has only recently become possible for users running on commodity hardware.

Idiolect integrates with Vosk, a state-of-the-art deep speech system with realtime models for various languages, which provides an open source Java API. Initially, we distributed a default model for the English language inside the plugin, but at the request of JetBrains to reduce bandwidth, we instead prompt the user to select and download a pretrained model from the Vosk website upon first installing the plugin.

Users may optionally configure a built-in TTS voice from the host operating system and a cloud-based speech recognition or synthesis service, with the caveat that web speech requires uninterrupted internet connectivity and introduces an additional 300-500ms of overhead latency depending on the user's proximity to the datacenter and other load factors.

[1] https://github.com/OpenASR/idiolect

IV. Intent Recognition

Once a spoken utterance is decoded as text, Idiolect must determine the relevant actions and entities needed to resolve the user's intent. Furthermore, it may need to consider the IDE context, i.e., the current editor state and command history, to resolve potentially ambiguous commands. For example, the command "open plugin menu" could refer to multiple different menus, depending on when and how it was invoked.

The IntelliJ Platform has over 10^3 possible actions. These actions are bound to keyboard shortcuts, menu items, and toolbar buttons. The user can also bind a voice command directly to an action, presuming the user already knows the action's identifier. Idiolect's default grammar was manually curated from the IDE action list, using the CamelCase identifier to generate a suitable description for intent recognition.

In keeping with the principle of configurability, we allow users to define custom grammars and bind utterances to actions using a context-free grammar. By providing an extensible DSL, users can bind their own command patterns using either a simple configuration file, or programmatically via the plugin API to handle more complex usage scenarios.

In some cases, the user may not know or recall the exact phrase to which an action they intend to perform was bound. Given an utterance that does not match any of the predefined grammars, the plugin will fall back to a language model (LM), a probabilistic model trained on a large corpus of text, which can be used to perform reasoning simple tasks like predicting the most likely intent from a list of alternatives. For example, the utterance "I want to edit foo.java" is more likely to match the command "open foo.java" than "execute foo.java".

A. *Recognition Dispatch*

Idiolect dispatches utterances to a series of recognizers using a tiered priority system. This system gives each recognizer the chance to match or pass on each utterance. Once a command is matched, the command is consumed and no subsequent recognizer will receive the dispatch.

The plugin first attempts to resolve a spoken utterance using an exact lexical match against a lexicon of predefined commands. Highest priority are those which control the plugin itself, enabling and disabling speech recognition.

User-defined commands are the next highest priority. These can be a lexical match, or more generally a sentence in a context-free language. Nonterminal parameters allow one to define to more complex instructions, such as "open the ⟨filename⟩[in ⟨project_name⟩]" or "jump to the ⟨nth⟩line".

If there is no exact match, we attempt to repair the phrase using an error-correcting parser. We describe this in Sec. V-E.

The recognizer of last resort is a large language model. Open ended commands which are unrecognizable by any of the previous approaches are dispatched to a service which matches the intent against a predefined lexicon. We can use a prompt, "What action is the most likely for the phrase …out of these actions: … ", then delegate to the appropriate action.

V. Barriers and Pathways to Usability

In our experience as plugin users and maintainers, usability challenges typically arise in a few key areas. We discuss some of these obstacles and our efforts to surmount them.

A. *User Onboarding*

Several users reported confusion when first installing our plugin. To address this issue, we added a wizard that guides users through their first install. Upon first installing the plugin, the user is greeted and prompted to download the Vosk model for recognizing their natural language of choice, which defaults to the system locale. Once unpacked, the model is stored in the plugin configuration directory, `/.idealect`. The user is next prompted to configure the properties file, and to bind a few voice commands. This is a one-time process, and users may reconfigure the plugin by opening the settings dialog at any time.

B. *Plugin Observability*

Failures can arise in many stages of the intent recognition pipeline. The best way to address this is by improving observability of the plugin, so that users can diagnose when an error occurs and learn how to avoid it in the future.

Failure to recognize is a common issue in intent recognition, the most common source of which is unrecognizability due to a transcription error by the ASR model, due to e.g., noise in the audio signal or poor recognition accuracy. To address this issue, we add a visual cue at the corner of the IDE that reports the phrase transcribed in realtime, as well as the action (if any) that was triggered on intent recognition.

C. *Command Discoverability*

We draw a distinction between capability discovery and intent recognition. Users previously familiar with the IDE capabilities may wish to invoke or bind a specific action directly, but may not know the command binding. The first and foremost way to improve discovery is through documentation, however keeping documentation in sync with capabilities can be challenging, and users are uninclined to read verbose documentation. To address this issue, we autogenerate documentation describing the action and their descriptions, by preprocessing the action list and assigning natural language descriptions from the IntelliJ Platform source code.

Another common scenario is when the user is unfamiliar with the IDE capabilities, and the action they wish to perform is not supported by the IDE directly. User awareness of IDE functionality is outside the scope of this plugin, however, many actions that users wish to perform have no associated binding or are compound actions. For example "delete the method named ⟨foo⟩" requires first resolving the method, selecting the body of the text and deleting it. These actions can be bound, but require a handcrafted recognition handler, and are generally quite brittle. We hope to improve scripting support for such actions in the future.

D. API Extensibility

Idiolect tries to anticipate users actions and give users additional action bindings. Some functionality, however, is best left implemeneted by downstream developers. In addition to end-user configurability, Idiolect is designed to be extensible by external plugin developers and can be used by other IntelliJ Platform plugins to programmatically define their own commands and recognition handlers. We provide a simple message passing API for plugins to communicate with Idiolect, and a DSL for defining custom grammars.

E. Error Recovery

Another common usability barrier is when transcription is accurate, but the utterance does not correspond to an actionable command, possibly due to stopwords verbal fillers or extraneous text which cannot be directly parsed. In short, if a given phrase, e.g., "open uh foo java" is received, we attempt to repair the utterance. At the user's discretion, or in case of ambiguity when the phrase has multiple plausible alternatives, we can either visually or verbally prompt the user to choose from a set of actionable phrases, e.g., "Did you mean ⟨a⟩open file foo.java, ⟨b⟩open folder foo/java, or ⟨c⟩something else?"

To address the issue of recognition errors, we apply Considine et al.'s (2022) work on Tidyparse, which supports supports recognition and parsing of context-free and mildly context-sensitive grammars, and computing language edit distance. Tidyparse implements a novel approach to error correction based on the theory of context-free language reachability, conjunctive grammars and Levenshtein automata. We use a SAT solver to find the smallest edit transforming a string outside the language to a string inside the langauge. When the utterance is one or two tokens away from a known command, we attempt to repair the utternace using error recovery.

VoskAPI is also capable of returning a list of alternate utterances, alongside a confidence score for each, which we use to determine if the user's utterance is sufficiently close to a known command in the list of alternates.

F. Build Reproducibility

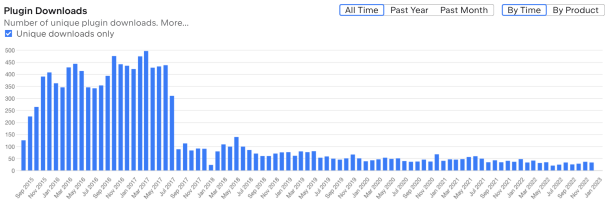
One of the challenges of managing a multiplatform audio project is ensuring reproducible builds. The IntelliJ Platform does not allow shipping OS-specific plugins, which requires building a single artifact. In the past, updates would result in breaking changes for Windows, Mac OS, or Linux, which are difficult to to debug or test on a development environment and only identified by user acceptance testing on those platforms.

To streamline the development process and mitigate the risk of platform-specific regressions in production, we set up an automated pipeline for testing and deploying the plugin. We use GitHub Actions to build the plugin for each release, and when a new commit is tagged with a release number and merged, a changelog is automatically generated from the intervening commit messages, then the plugin is signed and automatically uploaded to the JetBrains plugin repository. This allows us to quickly iterate on the plugin, test the plugin on multiple platforms, and to release new versions with assurance.

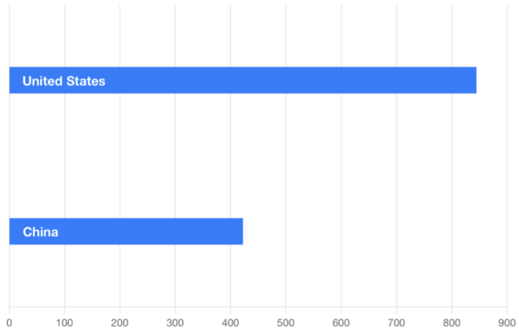
VI. Evaluation

We evaluated the plugin on a variety of supported tasks. Our primary means of evaluation was user downloads of the plugin over a five-year timespan. We also performed a survey of users, to determine how satisfied they were with the plugin.

VII. Results



Downloads of the plugin have stalled over time since its initial release in 2015. We hypothesize this is due to the lack of steady updates, resulting in incompatiblity with more recent versions of the IntelliJ Platform. Originally developed using the CMUSphinx speech recognition platform, we then performed an intervention by migrating to a more robust and technically capable ASR engine. Following a five-year hiatus, we rewrote the plugin from scratch, implemented the usability improvements described in Sec. V, published a new release under its current name in 2023 and are collecting usage statistics for the new plugin – though promising, the results of this ongoing experiment are inconclusive.



A careful inspection of demography indicates a large fraction of the plugin downloads originated from the People's Republic of China, indicating a substantial and potentially underserved programming demographic. A careful analysis suggests the need to support internationalization and localiation, an omission that we hope to remedy in a future release.

In addition, we conducted a preliminary experiment to evaluate recognition performance on synthetic speech. Using a set of English TTS voices (both male and female) included in MacOS Ventura 13.1, we synthesized 100 utterances from the predefined command list using the `say` command, transcoded a 16000 kHz PCM signed 16-bit little-endian audio recording using `ffmpeg`, transcribe the audio stream using VoskAPI to simulate a user-generated microphone recording, and report the average word error rate of three ASR models trained on US English speakers: `vosk-model-en-us-0.22` (0.30) `vosk-model-small-en-us-0.15` (0.28), and `vosk-model-en-us-0.22-lgraph` (0.35). Results on a per-voice and per-model basis are shown in Table **??**.

a	b	c	d	e	f	g	h	i	j	k
NaN	0.58	0.63	0.63	0.98	0.29	0.49	0.53	0.89		
NaN	0.63	0.66	0.67	0.98	0.35	0.54	0.57	0.88		
NaN	0.79	0.74	0.76	1	0.27	0.72	0.66	0.9		

VIII. Future work

In the future, we plan to conduct a thorough user study to better understand the use cases for the plugin. In particular, we hope to better understand the development habits of visually and motor-impaired users to improve accessibility, as well as broader support for other languages and dialects.

We also plan to conduct an evaluation of the plugin's intent-recognition capability and improve support for personalization, such as speaker adaptation and user-specific language models.

We also aim to improve the interaction mechanism by adding audiovisual feedback and modal voice commands. We would like to provide a more user-friendly configuration interface, possibly using an embedded DSL for defining dialog trees with a choose-your-own-adventure style modal dialog tree navigator. Futher integrating modal logic a la Hazel by adding a visual backpack and modal operators would be an interesting direction to take and one we hope to explore.

Currently, we only track download statistics, however to inform the development of these features and better address the needs of voice programmers, it would be helpful to collect minimal telemetry on user utterances and custom commands.

Recent progress in machine learning has enabled the use of large language models (LLMs) for a variety of tasks, including speech recognition, machine translation, and text generation. An LLM trained on a large corpus of natural language can be used to perform simple reasoning tasks, such as predicting the most likely intent from a given utterance, for example, the utterance "I want to edit foo.java" is more likely to be the command "open foo.java" than the command "execute foo.java". While LLMs are currently served on the cloud, recent efforts to compress and do inference on commodity hardware are ongoing. We predict in the next few years these models will soon be available on the edge, and can be used to perform real-time intent recognition.

Finally, we would like to use language models to rerank the most likely utterances, conditioned on a previous context of historical commands, where we attempt to locate the most likely intent matching the user's utterance, given the dictionary, context and a list of alternate utterances.

IX. Conclusion

In this work, we presented Idiolect, a plugin for the IntelliJ Platform that allows users to control the IDE using voice commands. We described the design of the plugin, and the challenges we faced while implementing it. We also presented the results of our evaluation, and discussed future work.

X. Acknowledgements

The first author wishes to thank his former colleagues Alexey Kudinkin and Yaroslav Lepenkin, who contributed to the plugin during the original JetBrains hackathon in 2015, and former manager Hadi Hariri for his advice and encouragement.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.