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

Jin Guo
McGill University
jguo@cs.mcgill.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 learn new words and phrases, and apply them in a variety of contexts relatively quickly. This is currently not the case for chatbots, which are often limited to a set of static commands and phrases defined at compile-time. This is a burden to bot-developers as well, who must anticipate user intent and write bindings for each new use case. On both sides, this presents a time-consuming and expensive challenge, resulting in an impedence mismatch between author expectations and user intent.

Voice coding allows users to quickly dictate their own commands and phrases without resorting to a Turing Complete language. For example, the user might say "whenever I say *open sesame*, open the settings menu", and the system will learn this command and open the settings menu whenever the user says "open sesame". Or "whenever I say *redo thrice*, repeat the last action three times". Or invoke a function in a scripting language, open a file, or other manually tedious chores.

This flexibility addresses a common usability voice UX design, where users are unable to express their intent in a way that the system understands. For example, a user may want to open a specific file, but the system only understands the command "open file". The user must then learn the system's command vocabulary, and then rephrase their intent in a way the system understands. This experience can be a frustrating one, and often results in users abandoning the system altogether.

Idiolect defines a default lexicon of phrases, but does not force users to learn them. Instead, we allow users to define their own commands and phrases on-the-fly, and then incorporate them into the system immediately. This shifts the burden of learning from the user to the system, which learns the user's idiolect, freeing users to express their intent in a way that is most natural to them.

Primarily, Idiolect observes the following design principles: (1) be natural to use, (2) be easy to configure, (3) get out of the user's way as quickly as possible. We believe that these principles are important for a system that is intended for developer-use, who are busy people and more than capable of configuring the system themselves. We also support motor-impaired users who have difficulty typing, or prefer to use a voice interface.

In this paper, we describe Idiolect, a dynamically reconfigurable system that allows users to teach the IDE new commands and actions on the fly, by either verbally or programmatically expressing the desired behavior. By targeting IDEs, whose users are already familiar with programming, commands that require complex instructions can be written programmatically, and then invoked on the fly with a keyword or phrase.

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 "the 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, do not consider IDE integration and 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 to be a transformative enabling technology for this problem.

III. Speech Models

Today, speech recognition, the problem of translating an audio waveform containing speech to text, is essentially a solved problem - one of the many speech recognition models would

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

work well enough for our purposes. However, we also wanted a solution with realtime offline speech recognition capabilities built on an open source deep speech pipeline, which has only recently been available on commodity hardware.

Idiolect integrates with Vosk, a state-of-the-art deep speech system with realtime models for various languages. VoskAPI[2] is open source system with a Java API, which we use. Initially, we distributed a default model for the English language inside the plugin, but at the request of JetBrains to reduce bandwidth, we instead provided a script to download the model from the Vosk website, and then prompt the user to download directly.

For TTS, we use the built-in voices from the parent operating system, via the jAdapterForNativeTTS[3] library.

We also allow users to integrate with various other cloud-based speech recognition and synthesis providers, with the caveat that online speech requires an internet connection and introduces an additional 300-500ms of overhead latency depending on the user's proximity to the datacenter and other load factors.

IV. Intent Recognition

Once speech is translated to text, Idiolect must determine the intent of an utterance, and extract the relevant actions and entities. Furthermore, we may need to consider the context, i.e., the current state of the editor and previous command history, to resolve potentially ambiguous commands. For example, the utterance "open the plugin menu" could refer to multiple different menus, depending on the context in which 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 actions to voice commands. However, to bind a new command, the user must first know the name of the action. Idiolect's default grammar was manually curated from this list, using the name to generate a suitable description for intent recognition.

Idiolect supports a variety of methods for acting on a user's utterance, which can be defined by string matching, a context-free grammar, and LLM-prompting. This framework forms an extensible DSL for the creation of custom patterns to match against transcribed speech, which can be defined by an end-user via a simple configuration file, or programmatically by a plugin developer to handle more complex usage scenarios.

The plugin first attempts to perform an exact lexical match, by attempting to resolve a given utterance against a predefined lexicon of commands and phrases. Idiolect next attempts to match the speech against a context-free grammar, which may contain named capture groups, used to extract parameters from the utterance. For example, we can match more complex patterns, such as "open the (?¡file¿.*) file in the (?¡project¿.*) project". This is a powerful tool for developers, who can define their own grammars to match against user utterances, and extract parameters from the utterance.

[2]https://github.com/alphacep/vosk-api

[3]https://github.com/jonelo/jAdapterForNativeTTS

This is the primary way to define and modify commands, and is typically the most reliable way to match a user-defined command. However, the user may not know or recall the exact phrase a given action they wish to perform was bound. In this case, if the utterance does not match any of the predefined lexicons or grammars, the plugin will attempt to match the utterance against a language model (LM), a probabilistic model trained on a large corpus of text which can be used to perform simple reasoning tasks like predicting the most likely command to execute. 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".

A. Command Prioritization

Highest priority commands are those that enable and disable speech recognition.

Then, user-defined commands.

Then the default commands from a plugin-wide grammar.

The recognizer of last resort are ChatGPT commands. We can use a prompt "What action is the most likely for the phrase "..." out of these actions: ..." and it will select top action as the command.

V. Error Recovery

Idiolect supports defining and recognizing context-free and mildly context-sensitive grammars. In keeping with the design principles of "parse, don't validate", we allow users to define their own grammars, and then bind them to actions. In many cases however, there are out-of-vocabulary (OOV) terms that cannot be parsed directly, or which the user has misspoken. When the utterance is one or two tokens away from an unambigious command, we attempt some error recovery.

We address the issue of recognition errors by incorporating Considine et al.'s (2022) work on Tidyparse. In short, if a given phrase "open foo.java" is uttered, but the word file is missing, we repair the string to "open file foo.java". Tidyparse implements a novel approach to error correction based on the theory of context-free language reachability, finite field arithmetic, 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.

Vosk 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.

Finally, we can use a language model to rerank the most likely utterances, conditioned on a previous context of historical commands. This is a form of error recovery, where we attempt to locate the most likely intent matching the user's utterance, given the dictionary, context and a list of alternate utterances.

VI. Large Langauge Models

Recent progress in language modeling has enabled the use of large language models (LLMs) for a variety of tasks, including speech recognition, machine translation, and text

generation. We can use a large language model to predict the most likely utterance given a sequence of words. 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 these models 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 intent recognition in real-time.

VII. User Onboarding

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.

VIII. Plugin Extensions

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 define their own commands. We provide a simple message passing API for plugins to communicate with Idiolect, and a DSL for defining custom commands.

IX. Build Automation

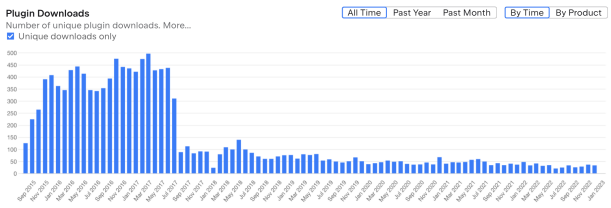
To streamline the development process, we set up an automated pipeline for building and deploying the plugin. We use GitHub Actions to build the plugin for each release, and when a new commit is tagged with a release number and merged, a changelog is automatically generated from the intervening commit 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 minimal effort.

X. Evaluation

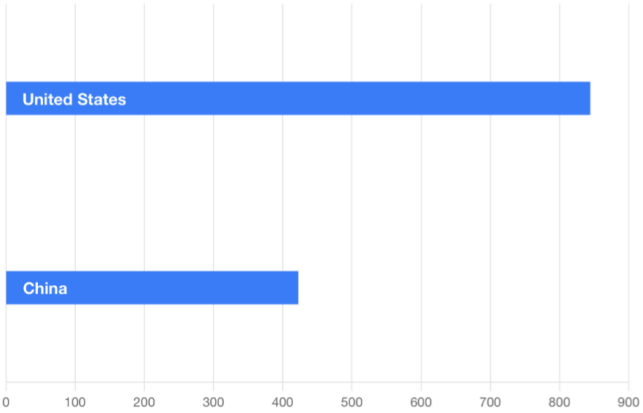
We evaluated the plugin on a variety of tasks, including opening files, creating new files, and running commands. We also evaluated the plugin on tasks outside the default grammar, such as creating new projects, and running Gradle tasks. Our primary means of evaluation was user downloads of the plugin over a five-year timespan. We also performed a survey of users, to determine how satisfied they were with the plugin.

We then performed an intervention, by switching to a new voice recognition engine, and are collecting data on the number of downloads of the new version – the results of this experiment are ongoing.

XI. Results



Downloads of the plugin have decreased over time since the plugin’s initial release. We conjecture this relates to the fact that plugin was not updated for several years, and thus incompatible with the latest versions of IntelliJ Platform.



A careful inspection of demography indicates a large fraction of the plugin downloads originate 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.

XII. Future work

In the future, we plan to conduct a thorough user study to understand the sociolectic background of the plugin’s users, and to gain a more complete understanding of its possible use cases. In particular, we hope to offer improved accessibility support to facilitate their development habits of visually and motor-impaired users, as well as broader support for other languages and dialects.

We also plan to conduct an evaluation and analysis of the plugin’s intent-recognition capability and to improve the support for personalization, such as speaker adaptation and user-specific language models. Finally we would like to provide a more user-friendly configuration interface, possibly using an embedded DSL for defining dialog trees.

To prioritize the development of these features, it would be helpful to collect telemetry to guide the development of the plugin and address the needs of voice programmers.

XIII. 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.

XIV. Acknowledgements

The first author thanks 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.