Iris: A Conversational Agent for Complex Tasks

Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, Michael S. Bernstein
Stanford University
{bchen45, jmendels, jbassen}@stanford.edu, {ethan.fast, msb}@cs.stanford.edu

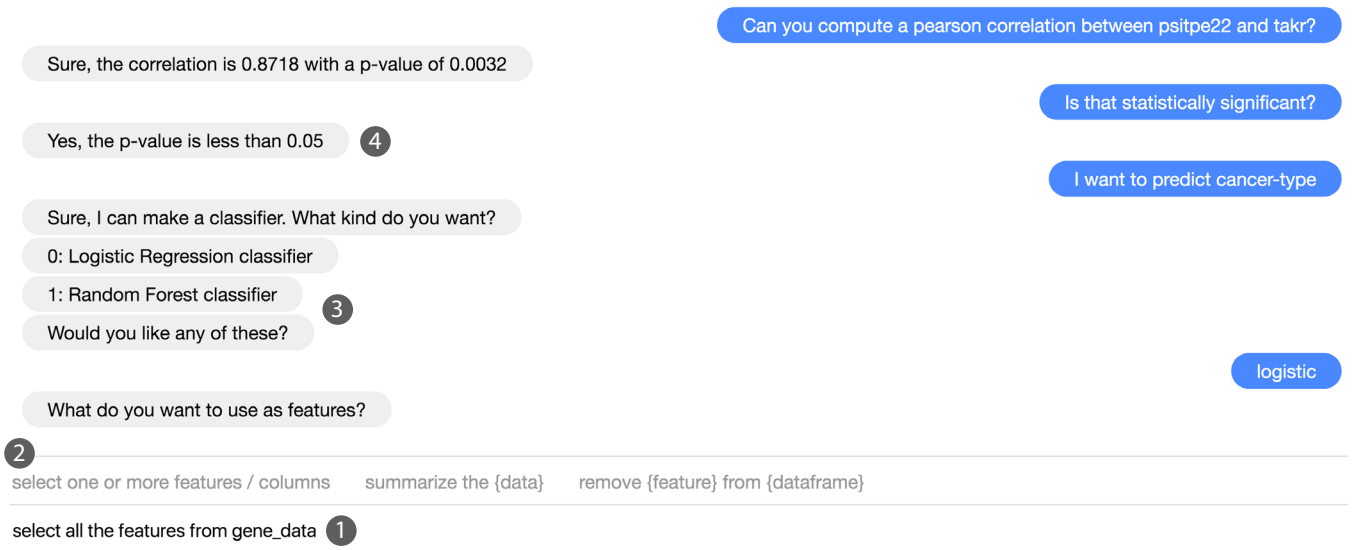

Figure 1: Iris allows users to combine commands through nested conversations to accomplish open-ended data science tasks. (1) Users interact with Iris through natural language requests and (2) the system responds with real-time feedback on the command the request will trigger. Once a command is triggered, Iris (3) converses with users to resolve arguments. When resolving arguments, users can (1) initiate a nested conversation via a new command, or (4) reference the result of previous conversation.

ABSTRACT

Today, most conversational agents are limited to simple tasks supported by standalone commands, such as getting directions or scheduling an appointment. To support more complex tasks, agents must be able to generalize from and combine the commands they already understand. This paper presents a new approach to designing conversational agents inspired by linguistic theory, where agents can execute complex requests interactively by combining commands through nested conversations. We demonstrate this approach in Iris, an agent that can perform open-ended data science tasks such as lexical analysis and predictive modeling. To power Iris, we have created a domain-specific language that transforms Python functions into combinable automata and regulates their combinations through a type system. Running a user study to examine the strengths and limitations of our approach, we find that data scientists completed a modeling task 2.6 times faster with Iris than with Jupyter Notebook.

Author Keywords

conversational agents; data science

CHI 2018, April 21–26, 2018, Montréal, QC, Canada.

https://doi.org/10.1145/3173574.3174047

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: Natural Language

INTRODUCTION

For decades, the promise of computers that communicate with us through natural language has been depicted in works of science fiction and driven research agendas in artificial intelligence (AI) and human-computer interaction (HCI). As early as 1964, Joseph Weizenbaum demonstrated how a computer program could hold open-ended conversations using a large set of pattern matching rules [46]. Terry Winograd later developed a more sophisticated program that could act upon natural language requests within a simplified blocks world [47]. In recent years, researchers have begun to apply these assistants to more complex tasks, such as data visualization workflows [41, 19].

Speak to today's agents as you would to a colleague or friend student, however, and it becomes clear that they have many limitations. In particular, agents are typically oriented towards executing standalone commands rather than complex conversation. Modern NLP techniques such as semantic parsing [45] and LSTMs [4] can decompose complex requests such as "run a t-test on the log-transform of x and y" into function composition over several independent commands. However, these techniques require large amounts of training data, making bootstrapping a new domain or command a difficult task that worsens combinatorially with the number of

combinable commands. As a result, commanding a modern conversational agent to run a t-test on the log transform of two variables is more likely than not to result in a response like "I don't know what you mean".

Linguistic theory suggests a complementary path towards conversational agents that support greater complexity. In daily life, people often build meaning through combinations of speech acts [16, 28]. Instead of asking an graduate student, "can you run a t-test on the log-transform of x and y", we might make the same request in stages via a nested conversation. For example, first we might ask, "can you run a t-test?" and when the student replies, "on what?", we can then clarify, "on the log-transform of x and y". This enables the same kind of command composition as a state-of-the-art NLP (and more expressivity than today's commercial agents such as Siri), but in a form that requires less training data and is more robust to errors. This is possible because instead of inferring an entire parse tree from a single statement, the agent only needs to correctly classify and execute each step of the conversation. The resulting conversation data and parse tree can then be used to train more advanced models.

We leverage this insight—that humans build meaning from combinations of atomic speech acts—to create a system that can nest and combine commands with far less training effort than modern deep learning systems. We draw on techniques from functional programming, such as first-order and partial functions, to create a domain-specific language (DSL) for authoring commands. Our system transforms Python functions into composable automata and regulates the set of possible automata compositions through a conversational type system. The result is a set of programming abstractions that enable an agent to compose one command through the argument of another via nested conversation, a form of function composition, or sequence commands through a referencing to some previous conversations, a form of assignment. From nearly one hundred atomic commands, our system enables thousands of composed commands.

We showcase this architecture in *Iris*, a conversational agent that helps users with data science tasks (Figure 1). Data science is a domain where complex tasks are often executed by non-expert programmers; yet these tasks are difficult to support with standalone commands [21]. To interact with Iris, you type natural language requests into a chat window (1.1) and receive real-time feedback on what command your request will trigger (1.2). When you enter a command, Iris converses with you to resolve its arguments (1.3), which you may populate by calling new commands, a form of conversational *composition* (1.1). You can also use the results of previous commands by storing them in named variables or referencing previous command results, forms of *sequencing* (1.4). If you are an expert user, Iris exposes an API that allows you to extend it with new commands.

The primary contributions of this paper include:

- An approach that allows agents to combine commands through nested conversation, inspired by linguistic theory.
- A DSL for transforming Python functions into composable commands that can be leveraged by a conversational agent.
- The Iris system: a conversational agent for data science.

CONVERSATION ANALYSIS THEORY

Linguistic models of human conversation suggest that agents can compose commands through nested conversations, as opposed to the more challenging task of inferring these compositions automatically from a single request. We use conversation analysis (CA), a theory that has been influential in sociolinguistics and discursive psychology [18, 14], to help us explain the human conversational strategies that allow agents to compose commands.

CA theory models all conversations through a basic unit called an *adjacency pair*: a pair of statements spoken by two conversational participants [18]. These pairs are typed with labels such as greeting-greeting or question-answer. More complex conversations can be described by joining adjacency pairs through expansions: In particular, *insert expansions* nest an adjacency pair within a conversation that allows one speaker to resolve other issues before continuing, for example, asking what time a World Cup game airs before answering a question about when to schedule a meeting.

Insert expansions are useful to agents because they can make conversation more expressive. Today's agents leverage a form of insert expansion called a *clarification request* [29]. Humans use these requests to clarify the meaning of previous statements. Agents often use them in a similar way, for example, asking for or confirming arguments before execution: "By Elena, did you mean Elena Ferrante?".

Unlike today's agents, Iris supports an insert expansion called the *dependent question*: a nested conversation where the meaning of one statement is grounded in a subsequent request [29]. For example, a request like "Who is going to the game tonight?", might depend a new conversation initiated by, "I don't remember". Iris supports dependent questions through command composition. For example, when running a command to compute the mean of some data, Iris might ask, "What array would you like to use?", and a user might answer, "I'd like to generate one from the normal distribution". After further conversation, the resulting array value will be used to compute the mean.

Finally, the linguistic idea of *anaphora* describes expressions of language that depend on previous expressions [37]. For Iris, anaphoric expressions are values produced by a previous command that are necessary to the execution of the present command, enabling command sequencing. Iris supports such expressions through named variables and a simple model of pronoun co-reference.

TODAY'S CONVERSATIONAL AGENTS

Iris is inspired by many existing conversational agents [17, 19, 5, 2] and we examine the interactions these agents support in Table 1. All systems have the ability to *execute* standalone commands, such as "set a timer" or "generate a random number"; *extract arguments* from a user query, for example, parsing the name Oyeyemi from "send Oyeyemi an email"; and *resolve arguments* by asking follow-up questions, for example, responding to the request "schedule a meeting" with "when would you like to set it?"

One important advance of Iris over existing agents is support for *composition* (i.e., nested conversations). For example, when Iris asks you a question, such as "what model should I use?" you can respond with a request that initiates a new conversation such as "make a logistic regression model". The result of that nested conversation will then pass back to the initial request. Other systems either do not support these interactions [2, 19], or only support them when they have been hard-coded into the agent via a dialog tree [5]. For example, when Siri asks, "What's the date of your event?", you can respond with some commands such as "when do I get back from Australia?", but not with others, such as "how about on Elena's birthday?".

	Example	Theory	Agents	Iris
Execute Command	Set an alarm for 12pm Okay, it is set.	Speech acts	✔	✔
Resolve Arguments	Set an alarm. When do you want to set an alarm?	Clarification Request	✔	✔
Extract Arguments	Make a reservation at *Oren's* for *5pm*. Okay, making the reservation.	NA	✔	✔
Compose Commands	Schedule a meeting with Elena. Sure, for when? The day I get back from Australia. Okay, setting the meeting.	Dependent Question	Hard-coded	✔
Sequence Commands	Who is my wife? Elena Ferrante. Call her. Okay, calling Elena.	Anaphora	Mostly Hard-coded	✔

Table 1: Interactions that today's conversational agents have been designed to support. Iris enables broader support for *composition*, calling one command within another command, and *sequencing*, using the results of previous commands in a new command.

Support for composition enables other new interactions that are absent from agents today, such as commands that take references to other commands as arguments (similar to first-class functions in programming languages), or commands that generate new commands (similar to partial functions). For example, Iris can combine "transform the dataframe" with "square each value", where the second command is captured as an argument of the first and applied repeatedly over the data.

Finally, Iris supports the *sequencing* of any of its commands (i.e., anaphora resolution). For example, you might ask Iris to "get the petal-length column from data.csv", then to "take the mean of that column''. A few existing agents support such open-ended sequencing [2, 19], but most support these interactions only when hard-coded.

In sum, most of today's agents support command combination through hard-coded logic. The enormous training difficulty of machine learning methods that infer command compositions automatically put these methods out of reach for many applications. By instead allowing a user to combine commands through nested conversations, agents can immediately unlock a much more complex set of tasks.

RELATED WORK

Iris is inspired by other dialogue systems that engage with complex tasks. Like Iris, these systems often represent commands as automata, and in some cases they allow users to combine commands interactively. PLOW and Ava, for example, allow users to sequence commands through variable assignment for complex tasks such as filling out a form on the web [2, 19], but do not support command composition through nested conversation. Similarly, Ravenclaw and Siri allow users to compose some commands through nested conversation, but only if the composition has been pre-defined in a logic tree [5]. A related class of methods such as semantic parsing or deep LSTMs also enable atomic command composition [45]. These methods require large amounts of training data, however, and do not support back-and-forth dialog with a user to clarify arguments or mistaken compositions. The key contribution of Iris over prior work is greater expressivity inspired by linguistic theory, allowing open-ended command combination through a simple statistical model. From a system design perspective, Iris also contributes a DSL that makes creating command automata as simple as writing and annotating a Python function.

Iris belongs to a line of HCI systems that map natural language to underlying functionality such as commands, code snippets, and APIs. Query-feature graphs provided an early foundation for these methods, connecting user requests with system commands in an image editing application [12]. Others have since extended this approach: for example, using word embedding models to connect user vocabulary with system keywords [1, 7] or programming syntax [38], and statistical language models to predict functions a user wants to call [11, 28, 8]. All of these systems solve the vocabulary problem [10] through statistical models that map natural language to the domain language of a system. While Iris takes a similar tack, it extends user requests through conversation, allowing users to combine atoms of functionality—through composition and sequencing–into complex commands.

Iris also draws on insights from tools designed to help with data science tasks [34, 44, 41]. Wrangler, for example, combines spreadsheet visualizations with natural language command descriptions to help users manipulate data [20]. Iris leverages similar natural language descriptions in the hints it displays as a user formulates a command. Other tools such as Burrito and Variolite are oriented more towards organizing data science code for reuse [15, 22]. Iris also aims to enable reusable code, but contributes a different perspective: by wrapping high-level functions in natural language, users can redeploy these functions in future conversations.

When designing for natural language interfaces, systems must manage the ambiguity of user language. DataTone and PixelTone provide guidance here, illustrating how systems can surface decisions about ambiguity [13] and constrain these decisions by direct manipulation [24] or other modalities [42]. Iris combines these ideas with an understanding

of how humans resolve ambiguity (e.g., through clarification requests [29]) to manage the execution of user commands.

Systems that interact with users through speech have unique psychological constraints [36], such as the tendency of users to anthropomorphize them [32]. To better design for these constraints, other work in HCI has examined how humans communicate through theories such as the language/action perspective [48] and applied these theories to agents [30, 43]. Such theories have centered around individual speech acts, which describe how language relates to the world [39]. Iris expands on this perspective by capturing the interleaving of multiple speech acts as described by CA theory [18]. A second class of work has turned a practical lens on the challenges of implementing speech interfaces, providing tools for rapid prototyping [23], triggering user queries [49], or extending the limits of machine reasoning with a crowd [6, 25, 26]. We similarly aim to provide a framework that allows others to design and bootstrap conversational agents.

SCENARIO

To motivate how combining commands through composition and sequencing can empower interaction with a conversational agent, we present a scenario in which we use Iris to replicate work analyzing linguistic signals of dogmatism (intractability of opinion) in social media [10]. For this scenario, Iris is seeded with a large set of statistical commands based on the *scipy.stats* and *sklearn* libraries, as well as commands that enable text analysis through lexicons [35, 9].

The original study collected dogmatism labels for Reddit posts through a crowdsourcing task. The dataset has two columns: *post*, the text of a post; and *score*, a dogmatism label between 0 (non-dogmatic) and 15 (dogmatic). We begin the scenario with the data loaded into the `dogmatism_data`, a dataframe indexed on these column names.

Linguistic Analysis of Dogmatism

To discover linguistic features associated with dogmatic posts, we first need to divide the posts into two populations: dogmatic and non-dogmatic. The original study considered posts in the highest quartile (top 25%) as dogmatic and those in the lowest quartile (bottom 25%) as non-dogmatic.

Basic Interactions with Iris

We type "quartiles" into Iris and the hint box fills with `compute quartiles for a {dataframe}`. We trigger that command and Iris asks, "What dataframe do you want to analyze?" The hint box fills with "dogmatism_data" because it is a variable in the environment that matches the type the command requires. We accept this suggestion, and Iris displays a spreadsheet view with the columns in the dataframe. We click on the "score" column to select it. Iris responds: "Q1 is from 2.0 to 7.0, Q2 is from 7.0 to 9.0, Q3 is from 9.0 to 12.0, and Q4 is from 12.0 to 15.0".

Complex Command Composition

Next we need to select the highest and lowest quartiles of posts: those with scores less than 7 (non-dogmatic) and those with scores greater than 12 (dogmatic). We type "filter dogmatism_data" which triggers the command `filter {dataframe} by {column}`. Iris outputs a spreadsheet view of the data and again asks us to choose a column to filter on. We

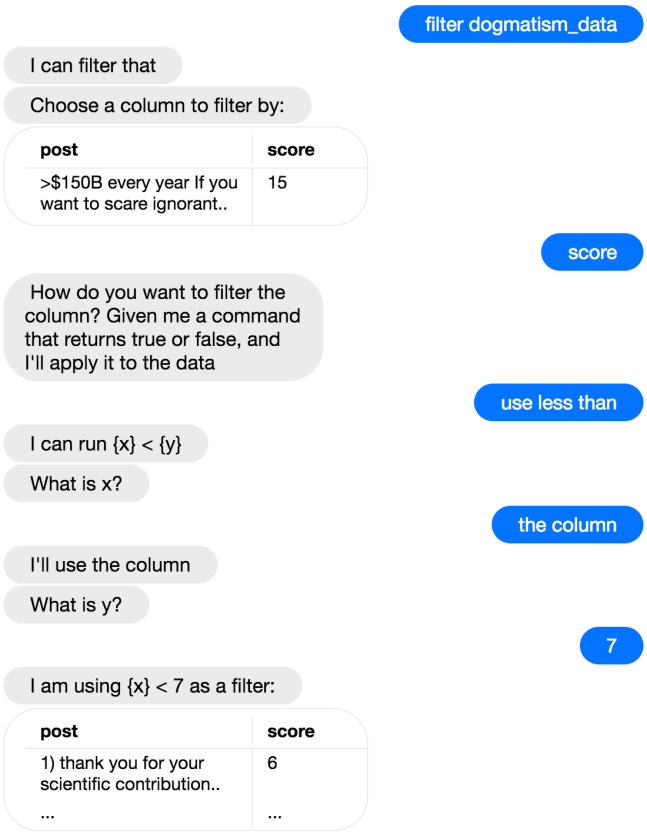

Figure 2: Composition allows Iris to support nested conversations. Here a user interacts with Iris to filter a dataframe to select rows where the score column is less than 7. Behind the scenes, Iris generates a new command using partial function generation to filter the data.

select `score`, and Iris asks, "How would you like to filter the column?", explaining that we can provide any command that takes a single argument and produces a `boolean` value. We say "use less than", which via composition triggers `{x} less than {y}`. Iris asks for the first argument (`x`) of the command, and we say "use the column" which triggers yet another command, `create reference to column variable`. Iris then asks for the second (`y`) and we say "7". Given the nested `create reference to column variable` command, Iris returns a partial function (`{x} less than 7`) that the `filter` command applies to each row of the dataframe, returning rows where the selected column is less than 7 (Figure 2).

The interaction above would not be possible in any other conversational agent today. First, *command composition* allows us to call another command within the `filter` function, then *partial command application* allows us to generate a new command dynamically from the `less than` command.

Conversational Type System

What if we had made a mistake and passed `filter` a command like `log base 10 of {x}`, which does not return a `boolean` value? In this case, Iris would evaluate the `log` command on the first row of data, returning a floating point number. This would fail a dynamic type check because `filter` requires a command that returns the type `boolean` and not `float`. Iris would then respond, "That filter argument requires a com-

mand that returns true or false, but your command returned a float", and repeat its original request. We could then try again, or exit the conversation via "quit". This type system provides a means of sanitizing inputs and composed commands before they are executed.

Saving and Sequencing Commands

We need to reuse the dataframe we just filtered, so we ask Iris to "save that as non_dogmatic_posts", which triggers `save {that} as {non_dogmatic_posts}`. A new variable named `dogmatic_posts` appears in the right side-bar. This interaction is possible because Iris can sequence commands, referencing the result of the previous command (via the keyword "that") in the current conversation. Through a similar interaction, we save posts with scores greater than 12 in `dogmatic_posts`.

Now we would like to test these dataframes of posts for differences in linguistic features. To replicate the original paper, we will use LIWC (Linguistic Inquiry and Word Count) [35], a popular tool for computational social science supported by Iris. LIWC analyzes text for signals across many linguistic categories, such as dominance, anger, or sentiment.

We tell Iris to "run an analysis using LIWC", which triggers the command `liwc analysis on {dataframe}`. We type "dogmatic_posts" and Iris outputs a spreadsheet view of the data ands asks us to choose a column. We select the `post` column, and Iris then runs the analysis and returns a dataframe indexed on word counts for each of LIWC's linguistic categories. We save these category scores in `dogmatic_liwc`, and then repeat this process for the content of non-dogmatic posts and save them in `non_dogmatic_liwc`.

Learning from User Language

We can now test for linguistic signals that are different between dogmatic and non-dogmatic posts. We type "run Mann-Whitney tests between the columns in dogmatic_liwc and non_dogmatic liwc", which triggers `statistical test {test} between {dogmatic_liwc} and {non_dogmatic_liwc}`.

Iris does not understand which statistical test we want (the `test` argument), so it responds, "Sure, I can run statistical tests between two dataframes. What test would you like to run?", along with a set of options that appear in the hint pane. We select Mann-Whitney U, and Iris connects this with "Mann-Whitney" in the original request, learning a new template for this command that it will remember in the future. Iris then executes the command to generate a dataframe of test statistics, which we save in `dogmatism_stats`.

Introspecting Iris Commands

The original study corrected these test statistics via the Holmes method. Has Iris already done so? We click on the `statistical test` function in the conversation pane to open up documentation in the right sidebar, which includes the command's source code and a description of what it does. Since we see that the statistics have not been corrected, we ask Iris to "apply Holmes correction to dogmatism_stats". After correction, we see dogmatic associations for *swearing*, *negative sentiment*, and *sexual language*, and non-dogmatic associations for *first-person pronouns* and *past tense*.

Visualizing Data

We would now like to investigate the relationships we just uncovered through data visualization. In particular, we can examine the relationship between the swearing LIWC category and dogmatism ratings through a scatter plot. We ask Iris to "make a scatter plot" using the LIWC data we already computed. By default the `scatter plot` command will plot a relationship between two columns in the same dataframe, but by using command composition, we can ask Iris to switch to a different dataframe to select the dogmatism score data for the y-axis (Figure 3). From the resulting plot, it is clear that while swearing often happens in both dogmatic and non-dogmatic posts, posts with high amounts of swearing are far more likely to be rated dogmatic.

Saving Conversations as Code

Finally, we would like to save our analysis as Python code to share with others. When conversing with Iris, the system both executes commands and dynamically constructs an abstract syntax tree (AST) behind the scenes that can be used

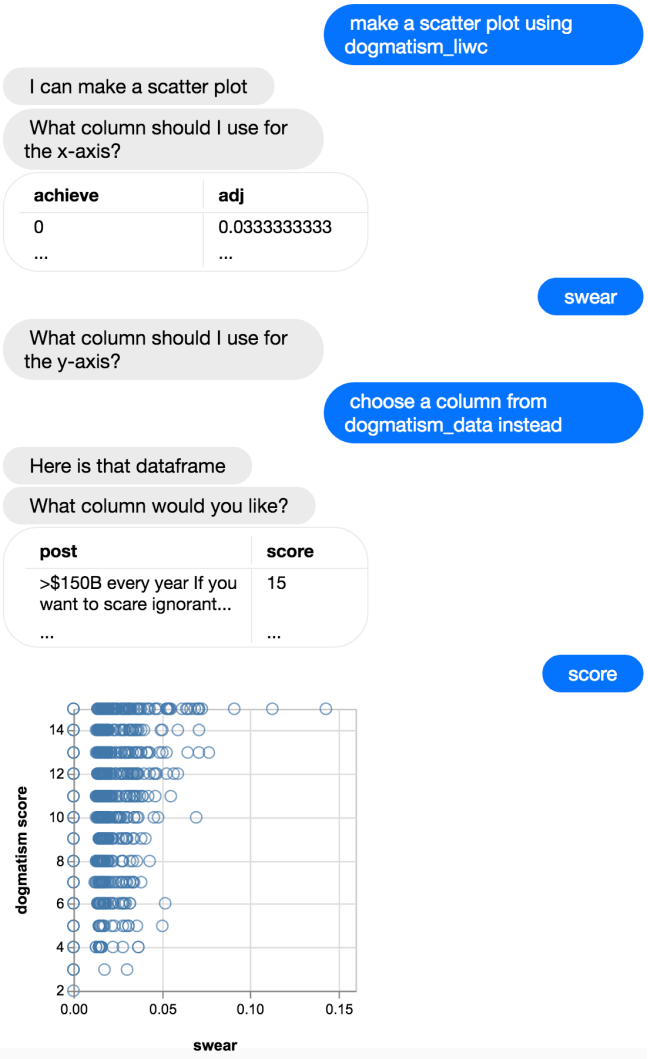

Figure 2: Iris displays a scatter plot for the relationship between sweating and dogmatism. By default, Iris interacts with users to visualize data from a single dataframe, but here we use command composition to select a column from a different dataframe for the y-axis data.

to save or re-run an interaction. We click on "export" in the right sidebar, which translates this AST into a standalone Python file that can be edited and run independently of Iris.

SYSTEM

Iris is a conversational agent that supports open-ended data science tasks. In this section, we describe how we built Iris, with emphasis on its compositional architecture, conversational type system, and API for command creation.

We present an overview of how Iris works in Figure 3. The entry point to interactions with Iris is a simple statistical model that maps user input onto commands. These commands require arguments that Iris either extracts through template strings or converses with a user to resolve. Users can answer an argument request by executing another command via a nested conversations (composition) or referencing a previous conversation (sequencing). This system is focused on a new approach to *combining atomic commands*, not command classification or argument extraction: other work in NLP will continue to improve on these core methods.

A Programming Model for Conversation

Iris draws upon many existing programming abstractions to support open-ended command combination. Previous dialog systems have used automata, or state machines [48], to model conversation. Automata provide useful scaffolding for back-and-forth interactions with a user, but they are not powerful enough to support nested conversations or commands that can reference and call other commands. We describe how we Iris has augmented traditional automata below:

Function application: Iris commands are functions wrapped by automata that interact with a user. For example, when arguments cannot be extracted from a user's request, automata will loop through clarification requests to request them. This basic functionality is shared by existing agents [19, 2].

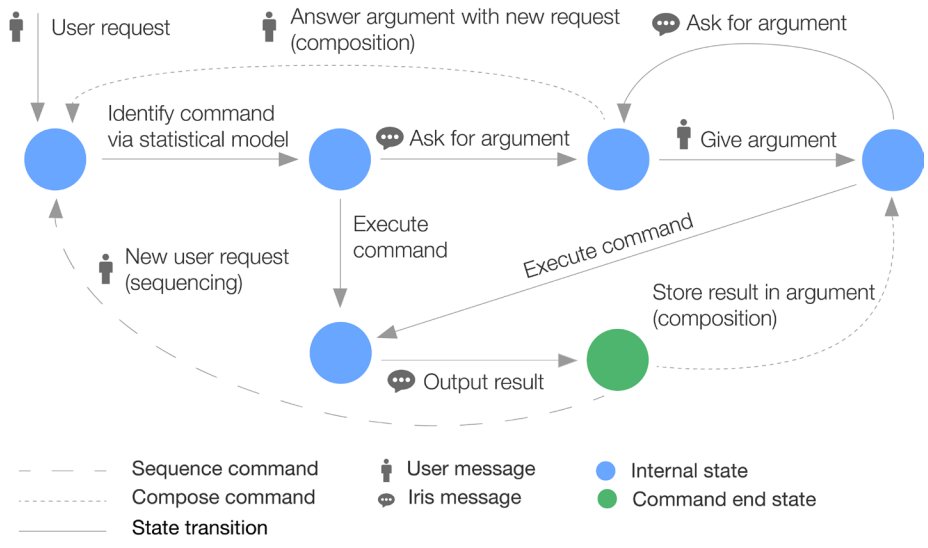

Figure 3: Iris allows users to combine commands by transforming them into automata that it can compose and sequence. Here we depict how these automata fit into the system architecture. *Composition* (short dash) allows comands to be called recursively within each other. *Sequencing* (long dash) allows commands to happen in series, referencing previous command results.

Function composition: Although Iris commands are automata, they can also be composed much like traditional functions. To support this, the states that handle argument requests can set their transitions dynamically to point to a new command, then loop back with a return value once the new command has executed. This functionality enables open-ended command compositions via nested conversation. This is a novle component of Iris, powered by other contributions below.

Memory: To share data between calls to Iris commands (i.e., sequencing), and to store data returned by argument requests and nested conversations (i.e., composition), automata need to read from and write to shared memory. To enable this, automata pass a dictionary between state transitions.

Scope: When composing two Iris commands with the same argument name, such as `x` in `add {x} {y}` and `subtract {x} {y}`, automata need scoping rules that prevent one command's argument from overwriting another's. To enable this, scoped commands store data in different namespaces.

Type System: To regulate what kinds of values can be passed to an Iris command, and which commands can be composed, some automata need type constraints over their transitions. This allows Iris to give commands a type signature that it can use to gracefully reject inputs at runtime.

First-class Functions: Some Iris commands take other commands as arguments. For example the `command` argument in `filter {dataframe} by {command}` requires a command that returns a `boolean` value when applied to its argument. To enable this, automata must be able to read and write other automata to and from shared memory.

Partial Functions: Iris can create new commands from commands it already knows. For example, a command such as `{x} greater than {y}` can be parameterized with arbitrary `y` values to create an infinite number of other single-argument functions, such as `{x} greater than 1`, `{x} greater than 2`, and so on. These partial functions are useful for filtering or transforming data, where you would otherwise need to register an intractable number of functions with Iris (one for each constant value to support). To enable this, Iris's automata support closures over argument bindings.

While existing work in conversational agents supports some of these concepts, such as function application or memory [2, 19], Iris is the first to support many others, such as composition, scope, and first-class functions. These concepts work together to allow users to combine commands via conversation.

The Structure of Iris Commands

Iris commands are building blocks that users can combine to accomplish complex tasks. These commands are defined by applying a

DSL to Python functions that extends them with conversational affordances such as: (1) what user requests trigger the function, (2) what types of arguments the function requires, and how Iris should interact with a user to resolve them, (3) how Iris can extract arguments from example requests, and (4) how Iris should communicate the return value to the user. In sum, Iris commands are functions annotated with conversational metadata.

We present an example Iris command in Figure 4. The *title* field describes how Iris will reference the command in conversation. The *examples* provide initial grounding for what user requests will trigger the command and how to extract arguments. The *argument_types* connect arguments with types that dictate how Iris should converse with a user to resolve them. The *explanation* function defines how the return value of a command will be presented to the user.

```
1  class PearsonCorrelation(IrisCommand):
2      title = "compute pearson correlation: {x} and {y}"
3      examples = [ "pearson correlation between {x} and {y}",
4                   "pearson correlation {x} {y}",
5                   "how are {x} and {y} correlated" ]
6      help_text = [ "..." ]
7      argument_types = {
8          "x": Array("Where is the first array to analyze?"),
9          "y": Array("Where is the second array?"),
10     }
11     def command(self, x, y):
12         from scipy.stats import pearsonr
13         return pearsonr(x,y)
14     def explanation(self, corr_pval):
15         corr = round(corr_pval[0],4)
16         pval = round(corr_pval[1],4)
17         return "Correlation of {} with p-value of {}".format(corr, pval)
18
19 pearsonCorrelation = PearsonCorrelation()
```

Figure 4: Implementation of an Iris command from a *scipy* function. Iris extends the function with conversational affordances, such as argument requests (lines 8-9) and an explanation (lines 14-17). These affordances translate into automata powered by the conversational DSL.

Iris commands inherit from a base class (`IrisCommand`) that transforms them into automata (Figure 3). An outermost automaton *extracts arguments* from a user's request and binds them to a command. This automaton transitions to another that *gathers missing arguments*, cycling through clarification requests with the user (driven by argument type). This in turn transitions to an *execution* automaton that looks up the argument bindings and runs the code.

The Conversational Type System

Iris's conversational type system provides a framework for sanity checking user input and resolving a command's arguments through conversation. For example, if Iris knows it needs an Integer, it can execute conversational logic to collect that value, such as, "What integer would you like to use for n?" or reject user input if the user provides a string.

Types in Iris consist of (1) logic that defines what Python values match the type, (2) a means of converting user input into a value of the type, such as a function to convert the string value "9" into an integer, (3) a clarification request that determines how Iris will interact with a user to resolve a type, and (4) a set of type converters that can transform values of non-matching types into the correct type, such as converting a floating point number into an integer.

Iris currently contains types for *integers*, *strings*, *arrays*, *dataframes* (multi-dimensional data with named columns), *models* and *metrics* (based on the sklearn API), and *plots* (based on the matplotlib API). These types all inherit from a base automata class (`IrisType`), so an expert user can add types to Iris without writing state transition logic. For example, the type definition for integers is only 8 lines of code.

A Statistical Model of User Requests

Iris connects user language with commands through a multi-class logistic regression model that is trained to predict commands from user language. Iris bootstraps training based on a few examples associated with each command, but as users enter new requests, the model is updated to incorporate them. With more data, it is possible to swap out this classifier with something more sophisticated, such as a deep neural network [40]. Iris does not contribute over existing NLP work for command classification: the classifier it uses is only important to enable the other, novel parts of its architecture.

Executing Requests via Iris Commands

To connect user requests with commands, Iris acts much like an interpreter in a traditional programming environment. Upon receiving a request, Iris calls its statistical model to determine which command the user wants. Iris then executes the desired command's automata, which handle argument extraction and clarification requests.

Extracting Arguments from Requests

The automata for argument extraction use an exact match procedure that aligns requests word-by-word with a set of templates and extracts values from matching words in the template so long as they can be converted into the correct type. As users enter requests it has not seen, Iris learns new templates for executing commands and updates its statistical model. When arguments cannot be extracted, these automata transition to others that handle clarification requests.

Given its modular architecture, Iris can easily be extended to support more advanced models for argument extraction (such as LSTMs) once we have more training data.

Composing Command Execution

Composition allows users to nest commands within each other through conversation (Figure 2A). For example, a user can compose one command to "transform the post column in dogmatism_data" with another to "take the absolute value".

To achieve composition, automata that handle clarification requests must distinguish between responses that are intended as primitive values (e.g., an integer, or the name of an array value) and those that correspond to new commands. These automata first use the conversion methods provided by their types to attempt to parse a user response into a value of the correct type. If this process fails, the automata will process the input as a new, composed command. Iris's interpretation of an input is transparent to the user and updated in real-time as a hint above the text field.

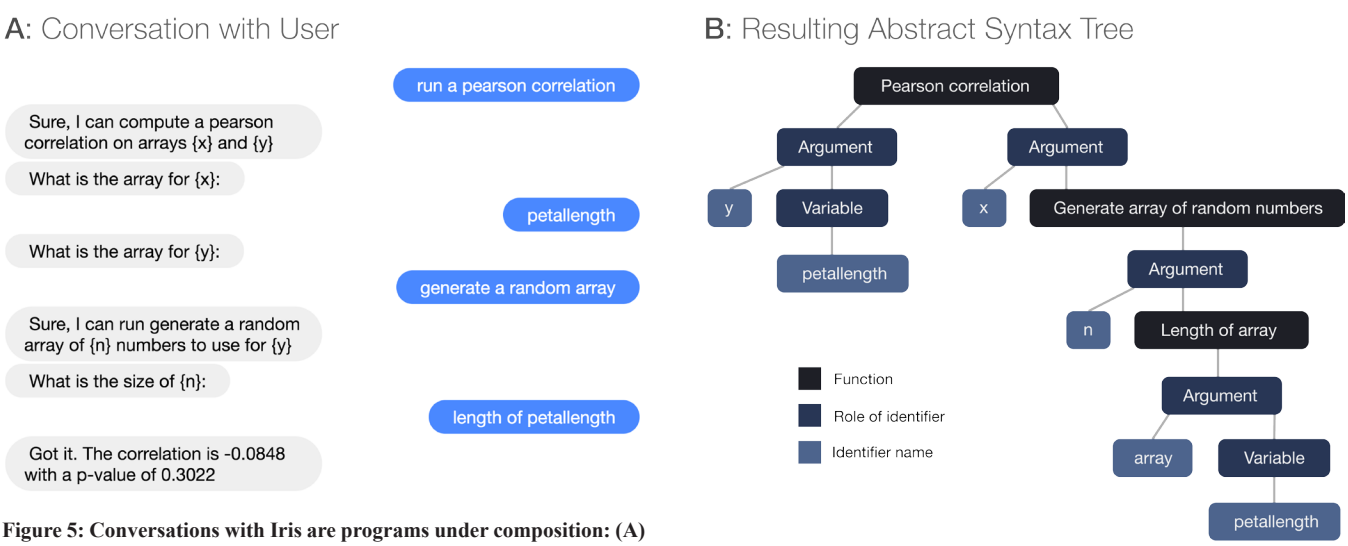

Figure 5: Conversations with Iris are programs under composition: (A) user conversation with Iris, (B) the abstract syntax tree that Iris builds at run-time, and (C) Python code generated by Iris from the AST.

To compose one command's automata (the child) with another's (the parent), the parent initializes the child with a new scope to prevent argument bindings from overwriting each other —for example, two commands that use the argument "x"—passes control of the conversation to the child, and adds a state transition from the child to the itself. After the child executes and transitions to the parent, the parent binds the child's result to the original argument in question, then continues its own execution (Figure 3).

Sequencing Command Execution

Sequencing allows users to reference the value of a previous command in their current interaction. To accomplish this, Iris saves the result of each command as it executes in a history variable that can be referred to in future conversations through pronoun keywords such as "this", "those", or "that". For example, after a command has just returned an array, users can ask Iris to "take the mean of that". This simple model works well in practice; in the future, it is possible to swap in more advanced co-reference models [33].

Iris also provides an internal API that commands can call to add new, named variables to the Iris runtime environment. We have used this API to create an general purpose command to `save {value} to {variable name}`, which can be issued to save conversation results for future use, such as in the request, "save that to array result", and then a follow-up, "tell me the variance of array result".

Transforming Conversations into Programs

Users can export conversations with Iris as a Python script to save and replicate their work. To accomplish this, Iris incrementally builds an abstract syntax tree (AST) that represents the current state of the conversation. When a user asks to export their conversation, Iris uses rules associated with each AST node type to compile the AST into a Python program. Concretely, Python source code for each relevant command is generated at the top of the program, sequenced commands are translated as variable assignment, and nested commands are translated as function composition (Figure 5C).

C: Conversion to Code

```
def pearson_correlation(x,y):
    from scipy.stats import pearsonr
    return pearsonr(x,y)
def random_array(n):
    import numpy as np
    return np.random.randn(n)
def length(arr):
    return arr.shape[0]
pearson_correlation(petallength, random_array(length(petallength))
```

The Iris User Interface

Users interact with Iris through a chat window that is augmented with hints and metadata about the current state of the system (Figure 1). Users enter requests into an input box (1.1) and receive real-time feedback about what command their re- quest will execute (1.2), which they can use to reformulate the request if necessary. Once a user has entered a command, Iris will begin a conversation in the chat window, asking questions if it needs information about a command's arguments (1.3). Users can respond to these questions with concrete values (for example, "Elena" or "2"), new commands that Iris should execute (1.4) or references to the results of previous conversations (1.5). The chat window has a right sidebar that provides information about the names, types, and values of existing environment variables.

Because Iris is designed for data science tasks, it must also display non-textual data. In particular, Iris outputs complex array and matrix data using *numpy*'s string formatting tools, pretty prints Python objects such as lists and dictionaries, and can embed images directly in the chat window.

The Iris user interface is built in JavaScript with React and Redux and communicates with a Python backend that runs the automata-based logic encoded by the conversational DSL. All backend and frontend components of the user interface are open source at http://github.com/Ejhfast/iris-agent.

EVALUATION

Can Iris help users accomplish data science tasks? What benefits and drawbacks does a conversational interface provide over programming? We ran a study to validate Iris's design,

and compare it to the high-level data mining API provided by the Python package sklearn. Following the study, we asked participants about their impressions of the system.

Method

Our study aims to compare Iris to how a data scientist would accomplish a predictive modeling task in Python. Study participants built and cross-validated a model to predict a flower's species based on measurements of the length and width of its petals and sepals [3], then reported on feature relationships by examining model coefficients. The task description was written by a third party expert who was not familiar with the training examples for Iris commands. We asked participants to complete the task as quickly as possible and measured the time it took them to complete the task as well as the correctness of their final output. Finally, we conducted semi-structured interviews, asking participants what they liked and disliked about Iris, and following up on any problems we noticed as they completed the study. These interviews lasted about ten minutes.

Experimental design: Participants completed the task using both sklearn (in a Jupyter notebook) and the Iris conversational interface, in random order to counterbalance learning effects. We chose to compare to sklearn due to the one-to-one correspondence between the commands in each tool (many Iris commands are built on sklearn).

Participants: eight people participated in the study. All were trained computer scientists with past experience in data science and sklearn, and none had used Iris prior to the study.

Data format: In the sklearn condition, we presented the flower data as a dictionary (with column names such as "sepal-length" as the keys). In the Iris condition, we presented these data as a dataframe (indexed on the same column names). Participants did not need to manipulate the data in either condition, besides selecting features to input to a model.

User instruction: In the sklearn condition, we provided a high-level explanation of how the library works, and URLs for all the sklearn methods that a participant would need to complete the task. We also answered any questions about sklearn in the course of the task. In the Iris condition, we explained how the interface worked, but we did not provide details about the commands that the participant would need, or answer questions about the purpose of Iris commands.

Results

Participants completed the task 2.6 times faster on average in the Iris condition (Figure 6). This effect was statistically significant under a Mann-Whitney U test (U_8=3.0, p<0.01). There was a small learning effect (participants were 1.31 times faster on average for the second interface), re-emphasizing the importance of our randomization, but the effect was not significant (U_8=26, p=0.28). All participants completed the task correctly in both conditions.

Participants approached the modeling task in different ways. Some participants worked backwards from a high level goal that was several steps in the future. For example, P1 asked Iris to "cross-validate" before they had created a model or selected features. When Iris then asked, "what model do you want to use?", P1 created it on demand though a nested conversation (made possible by *composition*). Other participants worked more incrementally. For example, P4 first selected features from the flower data and saved them in named variables, then referenced those variable names in conversation when creating a logistic regression model (made possible by *sequencing*). Most participants took a middle ground approach that leveraged both kinds of combination: for example, composing model creation and data selection, then sequencing that with a new request for cross-validation.

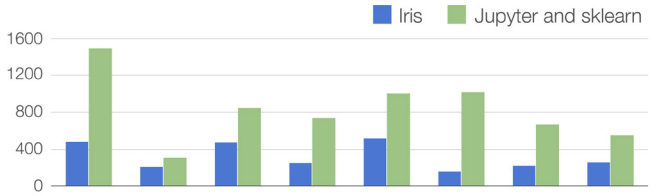

Figure 6: Study participants completed a data science task 2.6 times faster on average with Iris than with sklearn and Jupyter ($p < 0.01$). The y-axis describes time in seconds.

Participants also differed in the vocabulary they used to interact with Iris. For example, P1 asked Iris for a "logistic regression", while P3 asked to "build a classifier", and P2 asked to "select columns" while P5 asked to "select features". Sometimes users reformulated a request when nothing appropriate appeared in the hint box, as occurred when one participant attempted to access a column of data using the '.' notation common to Python objects.

While three participants interacted with Iris almost entirely through full natural language queries, most interacted via keywords, as if they were querying a search engine (for example, typing "logistic regression" to trigger `create a classification model`). When asked why they chose to converse this way, participants said it was faster, as the hint box indicated when keywords would trigger the correct command. Notably, keyword searches still allowed participants to build complex commands through composition and sequencing. In contrast, participants who interacted with Iris through full language queries (for example, "cross-validate model1 with accuracy and 10 folds") reported that they wanted Iris to extract a command's arguments automatically.

Advantages to Conversation

In interviews following the study, participants mentioned aspects of Iris that they liked and disliked. These aspects varied to some degree with a user's level of experience with sklearn. For example, less experienced users of sklearn found value in the *structural guidance* provided by conversation with Iris (for example, how APIs fit together). P1 said:

> "It took a while to remember how to thread together the components of the sklearn API. Like, first you need to initiate a model instance, then pass that to the cross-validation function—it wasn't totally obvious. But with Iris, once it knew I wanted to cross-validate, it walked me backwards though the steps, like giving it a model, what type of model, and so on."

In contrast, more experienced users thought Iris would *save time* as a wrapper for a set of functions they frequently call in smaller scripts. For example, P4 said:

> "For simple scripts, this is so much faster... I really like that you don't have to remember the name of the function you

want to call. All you need to do is say something close, like 'rename column' and it can figure out the rest."

Other participants commented on the *complexity* of what the system could accomplish. For example, P3 said:

"It's so cool that Iris can do such complex stuff, when legit companies out there have nothing this sophisticated."

Four participants expressed interest in using Iris for future data science work. One participant offered to connect us with a colleague in the Psychology department who was teaching an introductory statistics class as applied through R. "Iris would be so much better," P4 said.

Challenges for Conversation

We also asked participants what they found challenging about Iris's conversational interface. One common view was that the expressivity of Iris presented more *opportunities for mistakes*. For example, P2 said:

"It's great that Iris can combine anything together, but that creates more opportunities for something to go wrong."

Iris's conversational type system can prevent many common user errors, such as combining commands that return incompatible types, providing guardrails on user data entry that encourage exploration and experimentation [31]. However, as the complexity of conversations increased, we noticed that participants occasionally got lost when dealing with many layers of sub-conversations. Visual cues such as tab indendation might help add clarity to these interactions.

Similarly, other kinds of mistakes can emerge from miscommunication between a system and user about what the system is actually doing. Along these lines, some participants expressed the concern that a natural language based system should be *transparent about the operations it is executing*, especially for data science work. P5 said:

"Say I have some data that's not normally distributed, and I ask Iris to do a t-test. How do I know it's doing the right one?"

While Iris repeats back the commands it is executing, this is not always enough to give users a clear sense of what is happening. In response, we have added a feature to Iris that allows users to inspect a command's underlying Python code. Iris code is largely written using high level APIs such as scikit, so it is possible to quickly inspect the constituent functions and determine what methods are being executed.

Other participants mentioned the *vocabulary problem* as a potential challenge. For example, P4 said:

"Everything worked for me, but I can imagine another user entering the wrong words and getting stuck."

The vocabulary problem has long been an issue for systems based on natural language. As Iris gains more users and language data, we aim to address this problem more formally by learning from logs of mistakes [16]. In the course of the pilot study, three participants entered requests that triggered the wrong command. For example, "make model" triggered "create a regression model" instead of "create a classification model"; and for two participants, entering the name of a column did not resolve to a command to extract that column from a dataframe. While these mistakes were foreshadowed by text in the system's hint box, users ignored or misinterpreted these hints, and future versions of the Iris might emphasize them further.

Finally, several participants suggested that *text is not always the best medium* for communicating. For example, P2 said:

"There are certain things, like selecting data columns, that I'd prefer to do by clicking on things."

Prior work has demonstrated strong results with mixed modality interfaces [24]. Such feedback drove our decision to include a spreadsheet view of dataframes in Iris, and we plan to incorporate other modalities in the future. Manipulating and transforming data, for example, can often be enhanced by interactive visualizations, and we plan to explore how such visual feedback might augment user conversation.

LIMITATIONS AND FUTURE WORK

Here we discuss challenges to overcome as Iris's user base and range of functionalities grow.

First, as Iris expands to support many more commands, interpreting user language may become more difficult. The system currently supports 95 atomic commands with a manually authored dataset of examples (roughly 5 examples per command). How accurate will command classification become as the set of commands grows larger? How many user examples are necessary to support a new command among a library of thousands of others? Addressing these concerns will become more important as we deploy Iris in the wild.

Second, Iris creates AST representations of conversations with users and so has the ability to save these recorded programs, which may combine multiple commands. This ability connects with work in programming by demonstration [27], and offers the potential for users to create complex, reusable workflows through natural language. Saving ASTs for reuse presents a usability challenge, however. For example, how should Iris ask a user which parameters in the AST are arguments, and which should be captured as constants? And is it be possible to mine useful higher-level commands by analyzing the ASTs of hundreds or thousands of users? We aim to explore these questions as Iris grows.

Finally, Iris enables exploratory and interactive data analyses, as you might conduct today in R or a Jupyter notebook. This is distinct from other data science work that requires enormous datasets, where training a model may take hours, days, or even weeks. To run these models, researchers typically setup and debug long-running pipelines of commands, which is not something Iris is currently designed to do. As Iris expands to allow users to create and save workflows though PBD, such pipelines may be more feasible to run.

CONCLUSION

In this paper, we show how conversational agents can draw on human conversational strategies to combine commands together, allowing them to assist us with tasks they have not been explicitly programmed to support. We showcase these ideas in Iris, an agent designed to help users with data science and machine learning tasks. More broadly, our work demonstrates how simple models of conversation can lead to surprisingly complex emergent outcomes.

REFERENCES

1. Adar, E., Dontcheva, M. and Laput, G. CommandSpace: modeling the relationships between tasks, descriptions and features. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, ACM, 2014.
2. Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M. and Taysom, W. Plow: A collaborative task learning agent. 2007.
3. Anderson, E. The species problem in Iris. In *Annals of the Missouri Botanical Garden*, 1936.
4. Berant, J., Chou, A., Frostig, R. and Liang, P., Semantic Parsing on Freebase from Question-Answer Pairs. In *EMNLP*, 2013.
5. Bohus, D. and Rudnicky, A. The RavenClaw dialog management framework: Architecture and systems. In *Computer Speech & Language*, 2009.
6. Cranshaw, J., Elwany, E., Newman, T., Kocielnik, R., Yu, B., Soni, S., Teevan, J. and Monroy-Hernández, A. Calendar. help: Designing a Workflow-Based Scheduling Agent with Humans in the Loop. In *CHI*, 2017.
7. Fast, E., McGrath, W., Rajpurkar, P. and Bernstein, M. Augur: Mining Human Behaviors from Fiction to Power Interactive Systems. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ACM, 2016.
8. Fast, E., Steffee, D., Wang, L., Brandt, J. and Bernstein, M. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, ACM, 2014.
9. Fast, E., Chen, B. and Bernstein, M. Empath: Understanding topic signals in large-scale text. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ACM, 2016.
10. Fast, E. and Horvitz, E., Identifying dogmatism in social media: Signals and models*, In EMNLP*, 2016.
11. Fast, E. and Bernstein, M. Meta: Enabling Programming Languages to Learn from the Crowd*,* In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ACM, 2016.
12. Fourney, A., Mann, R. and Terry, M. Query-feature graphs: bridging user vocabulary and system functionality*,* In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, ACM, 2011.
13. Gao, T., Dontcheva, M., Adar, E., Liu, Z. and Karahalios, K. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ACM, 2015.
14. Gee, J. An introduction to discourse analysis: Theory and method. Routledge, 2014.
15. Guo, P. and Seltzer, M. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. *In TaPP*, 2012.
16. Hartmann, B., MacDougall, D., Brandt, J. and Klemmer, S. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2010.
17. Hauswald, J., Laurenzano, M., Zhang, Y., Li, C., Rovinski, A., Khurana, A., Dreslinski, R., Mudge, T., Petrucci, V., Tang, L. and Mars, J. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers*,* In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, New York, NY, USA, 2015.
18. Hutchby, I. and Wooffitt, R. Conversation analysis. Polity, 2008.
19. John, R., Potti, N. and Patel, J. Ava: From Data to Insights Through Conversations. *In CIDR*, 2017.
20. Kandel, S., Paepcke, A., Hellerstein, J. and Heer, J. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2011.
21. Kandel, S., Paepcke, A., Hellerstein, J. and Heer, J. Enterprise data analysis and visualization: An interview study. In *IEEE Transactions on Visualization and Computer Graphics*, 2012.
22. Kery, M., Horvath, A. and Myers, B. Variolite: Supporting Exploratory Programming by Data Scientists*. In CHI*, 2017.
23. Klemmer, S., Sinha, A., Chen, J., Landay, J., Aboobaker, N. and Wang, A. Suede: A Wizard of Oz Prototyping Tool for Speech User Interfaces. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, 2000.

24. Laput, G., Dontcheva, M., Wilensky, G., Chang, W., Agarwala, A., Linder, J. and Adar, E. Pixeltone: A multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2013.

25. Lasecki, W., Wesley, R., Nichols, J., Kulkarni, A., Allen, J. and Bigham, J. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, ACM, 2013.

26. Lasecki, W., Thiha, P., Zhong, Y., Brady, E. and Bigham, J. Answering visual questions with conversational crowd assistants. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ACM, 2013.

27. Li, T., Azaria, A. and Myers, B. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *CHI'17*, 2017.

28. Little, G. and Miller, R. Keyword programming in Java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, 2007.

29. Lupkowski, P. and Ginzburg, J. A corpus-based taxonomy of question responses. In *IWCS 2013 (International Workshop on Computational Semantics)*, 2013.

30. Maes, P. Agents that reduce work and information overload. In *CACM*, 1994.

31. Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. The Scratch Programming Language and Environment. In *Trans. Comput. Educ.*, 2010.

32. Nass, C. and Brave, S., Wired for speech: How voice activates and advances the human-computer relationship, MIT press Cambridge, MA, 2005.

33. Ng, V. Supervised noun phrase coreference research: The first fifteen years. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, Association for Computational Linguistics, 2010.

34. Patel, K., Bancroft, N., Drucker, S., Fogarty, J., Ko, A. and Landay, J. Gestalt: integrated support for implementation and analysis in machine learning. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, ACM, 2010.

35. Pennebaker, J., Francis, M. and Booth, R. Linguistic inquiry and word count: LIWC 2001. *Mahway: Lawrence Erlbaum Associates*, 2001.

36. Porcheron, M., Fischer, J. and Sharples, S. "Do animals have accents?": talking with agents in multi-party conversation. *In CHI*, 2016.

37. Reinhart, T. The syntactic domain of anaphora. Massachusetts Institute of Technology, 1976.

38. Rong, X., Yan, S., Oney, S., Dontcheva, M. and Adar, E. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ACM, 2016.

39. Searle, J. Speech acts: An essay in the philosophy of language. Cambridge university press, 1969.

40. Serban, I., Sordoni, A., Bengio, Y., Courville, A. and Pineau, J. Building end-to-end dialogue systems using generative hierarchical neural network models. In *arXiv preprint arXiv:1507.04808*, 2015.

41. Setlur, V., Battersby, S., Tory, M., Gossweiler, R. and Chang, A. Eviza: A Natural Language Interface for Visual Analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ACM, 2016.

42. Suhm, B., Myers, B. and Waibel, A. Multimodal error correction for speech user interfaces. In *ACM transactions on computer-human interaction (TOCHI)*, 2001.

43. Sun, M., Chen, Y. and Rudnicky, A. An intelligent assistant for high-level task understanding. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*, ACM, 2016.

44. Talbot, J., Lee, B., Kapoor, A. and Tan, D. EnsembleMatrix: Interactive Visualization to Support Machine Learning with Multiple Classifiers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, 2009.

45. Wang, S., Liang, P. and Manning, C. Learning Language Games through Interaction. In *CoRR*, 2016.

46. Weizenbaum, J. ELIZA—a computer program for the study of natural language communication between man and machine. In *Communications of the ACM*, 1966.

47. Winograd, T. and Flores, F. Understanding computers and cognition: A new foundation for design. Intellect Books, 1986.

48. Winograd, T. A language/action perspective on the design of cooperative work. *In Human–Computer Interaction*, 1987.

49. Xu, G. and Lam, M. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. 2017.