

From Data to Insights through Conversation

by

Rogers Jeffrey Leo John

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: May 19, 2021

This dissertation is approved by the following members of the Final Oral Committee:

Jignesh M. Patel, Professor, Computer Sciences

Mark Craven, Professor, Biostatistics and Medical Informatics

Theodoros Rekatsinas, Assistant Professor, Computer Sciences

Paris Koutris, Assistant Professor, Computer Sciences

© Copyright by Rogers Jeffrey Leo John 2021
All Rights Reserved

*Dedicated with love to my son Viyan Meero, my wife Meena, my mother Lalitha,
and my father Leo John for being my greatest inspiration and support.*

Acknowledgments

Ph.D. has been a transformative experience for me and has helped me grow both professionally and personally. Jignesh, you continue to inspire me with your impeccable work ethic, unbridled enthusiasm, and curiosity. Thank you, for all the opportunities and for pushing me towards becoming the person I am today. I look forward to our adventures in DataChat.

This journey has been incredibly challenging. I would not have gotten here without the support of my parents. Thank you very much, Dad and Mom, for encouraging me to dream beyond what I thought I could do. I've put you through a lot, and yet you've never failed in your support. I'm incredibly grateful for all the sacrifices you've made for me.

I've spent half a decade in Madison. I never once felt like I was away from my home. Thanks to my friends Adalbert Gerald Soosai Raj, Kaviya Lakshmipathy, Ramnatthan Alagappan, Aishwarya Ganesan, Harshad Deshmukh, and Supriya Hirurkar. I'll cherish our game nights, potlucks, tennis matches, and all the fun conversations.

Thank you, Ramakrishnan Durairajan and Thanumalayan Sankaranarayana Pillai, for the insightful conversations about life, research, and everything in-between. Your advice helped me navigate my Ph.D. I'm glad that I got to know you early in grad school.

Many thanks to Kirthanaa Raghuraman, Arjun Gurumurthy, Manoj Nagarajan, Ashok Marannan, Gopalakrishnan Uthamabikshandeswaran, Ameya Raul, and Siddharth VinothKumar for all the fun times and for

being test subjects for my research.

Thank you, Dilip Venkatesh, Perianayaki Krishnan, and AnandaPadmanabhan, for being one of the very few who stood by me when it mattered the most. I'm very grateful for all your help and support.

I'm very thankful to my undergraduate professor and mentor, Dr. T. Hamsapriya. Your encouragement and dedication to helping students learn have left an indelible mark on me. I'd also like to thank Dr. R. Arumuganathan and Dr. R. Nadarajan from my undergraduate university for providing me with a new perspective and approach for learning Computer Science.

I am deeply indebted to my professors Dr. Rebecca J. Passonneau, Dr. Noemie Elhadad and Dr. Sambit Sahu at Columbia University for the research opportunities, mentorship, and advice.

I'd also like to thank my colleagues (past and present) at DataChat. I've learned a lot working together with you. I look forward to continuing our collaboration.

I am very thankful to my graduate coordinator, Angela Thorp, for her tremendous support throughout this journey. Thank you, Angela, for being a good friend and family in Madison.

Meena: You are my friend, partner, motivator, and my pillar of support. I'm very fortunate and thankful for your boundless love and your steadfast support. I would have given up on my Ph.D. if not for your constant encouragement. Thank you very much for giving me the gift of being a father to our wonderful son Viyan. Thank you for standing by me every minute of this journey. We finally made it!

Contents

Contents	iv
List of Tables	vii
List of Figures	viii
Acronyms	x
Abstract	xi
1. Introduction	1
1.1 <i>The Data Science Workflow</i>	1
1.2 <i>Characteristics of a Data Science Workflow</i>	4
1.3 <i>Motivation</i>	5
1.4 <i>Approach</i>	6
1.5 <i>Contributions</i>	7
1.6 <i>Comparison with other NL-to-task mapping approaches</i>	8
1.7 <i>Future Work</i>	10
1.8 <i>Scope and Limitations</i>	10
2. Ava: An Intelligent Conversational Agent for Data Analysis	12
2.1 <i>Related Work</i>	12
2.2 <i>System Design and Architecture</i>	14
2.2.1 <i>Client-Server Jupyter Notebook Design</i>	16

2.2.2	Control Flow Architecture	16
2.2.3	Controlled Natural Language (CNL)	20
2.2.4	Artificial Intelligence Markup Language	21
2.2.5	Task Template Repository	22
2.2.6	Conversation Storyboard	22
2.2.7	Salient Aspects	24
2.3	<i>Experiments</i> 25	
2.3.1	Single-Subject Trial	25
2.3.2	Multiple-Subject Trials	26
2.4	<i>Results</i> 28	
2.5	<i>Conclusions</i> 32	
3.	DataChat: Ava Redesigned 33	
3.1	<i>Flexible and Robust CNL</i> 33	
3.1.1	Terminology	34
3.1.2	Pattern Language	34
3.2	<i>CNL Engine</i> 39	
3.2.1	Intent Classification	40
3.2.2	Entity Extraction	40
3.3	<i>Auto-suggest</i> 42	
3.4	<i>Skills as a formalism of task templates</i> 43	
3.5	<i>Modified Client-Server Architecture</i> 44	
3.5.1	The DataChat App Ecosystem	44
3.5.2	Chat Client	47
3.5.3	DataChat Worker	47
3.5.4	Data Stores	48
3.5.5	Transactions Over Object Stores	49
3.5.6	Publish-Subscribe Communication Mechanism	49
3.5.7	DataChat Scheduler	50
3.6	<i>Save and Replay Mechanism</i> 51	
3.7	<i>DataChat as a Platform</i> 52	

3.8	<i>Scalability and Deployment</i>	52
3.9	<i>Conclusion</i>	53
4.	DataChat: Ava Revisited	54
4.1	<i>ML Pipeline</i>	54
4.1.1	ML Pipeline Advantages	55
4.2	<i>Design Principles</i>	57
4.2.1	Execution Model	59
4.2.2	Constituents of the ML Pipeline Graph	59
4.3	<i>AutoML in DataChat</i>	61
4.3.1	Feature Transformations	64
4.3.2	Class Imbalance Handling	65
4.3.3	Explanations	66
4.4	<i>Experiments</i>	72
4.4.1	Results	73
4.4.2	Productivity Improvement	73
4.4.3	Qualitative Feedback	82
4.4.4	Other Observations	84
4.5	<i>Conclusion</i>	87
5.	Conclusion and Future Work	89
5.1	<i>Future Work</i>	91
5.1.1	Model Management	91
5.1.2	Developing tools for collaborative analytics	91
5.1.3	Multi-Modal User Interface for Task Specification	92
5.1.4	Uncontrolled NL to SQL (NL2SQL)	93
	Bibliography	94

List of Tables

4.1	Qualitative feedback from the participants of the user study	83
5.1	Flesch reading ease scores for DataChat's AutoML CNL . . .	93

List of Figures

1.1	The Data Science (DS) workflow for Example 1.1. Tasks are shown in rectangles, and meta-tasks in ovals. The pipeline is highlighted in dotted blue boxes.	2
1.2	The Data Science Venn diagram	5
2.1	A timeline of an actual Ava conversation for the data science task in Example 1.1.	15
2.2	Ava in action. The right panel is the chat window. The conversation between the user and Ava is carried out in this chat window. Actual code is created and run in the left panel using the regular Jupyter Notebook interface.	17
2.3	Overview of Ava’s control flow architecture.	18
2.4	The storyboard powering Ava, excerpted and truncated for brevity.	23
2.5	Distribution of the time taken by participants to complete the first model.	28
2.6	Distribution of the time taken by participants to complete the first model in each user group.	30
2.7	Distribution of unbiased task completion times.	30
3.1	Auto-suggest in DataChat	43
3.2	The DataChat microservices architecture.	45
3.3	The DataChat app ecosystem.	46

4.1	Multiple analysis without ML pipelines	56
4.2	Multiple analysis with ML pipelines.	57
4.3	Feature importance metrics generated by DataChat's AutoML for the churn prediction task	67
4.4	Data explanations generated by DataChat's AutoML module for the churn prediction task	69
4.5	Steps for generating the data explanation in Figure 4.4 . . .	70
4.6	Pipeline report generated by DataChat's AutoML framework provides transparency into the model training process	71
4.7	Distribution of time taken by participants to complete the first ML model using DataChat and Python.	74
4.8	Distribution of the number of models trained by participants using DataChat and Python.	75
4.9	Distribution of the number EDA questions attempted using DataChat and Python.	76
4.10	Distribution of the time taken to answer EDA questions using DataChat and Python.	77
4.11	Comparison of CNL errors made by users during the initial and later stages of the user study. Users first solved the EDA task followed by the modeling task.	78
4.12	Distribution of the time taken by the task groups to train the first model.	80
4.13	Breakdown of the errors made by participants in the EDA task.	81
4.14	Distribution of f1-scores on the test dataset.	85
4.15	Comparison of f1-scores of the best model for each user trained using DataChat and Python	86

Acronyms

CNL Controlled Natural Language

DS Data Science

ML Machine Learning

NL Natural Language

Abstract

Enterprises increasingly employ a wide array of tools and processes to make data-driven decisions. However, there are significant inefficiencies in the enterprise-wide workflow that stem from the fact that business workflows are expressed in Natural Language (NL), but the actual computational workflow must be manually translated into computational programs. This thesis presents an initial approach to bridge this gap by targeting the Data Science (DS) component of enterprise workflows. In this initial approach, we propose using a conversational agent to allow data scientists to assemble data analytics pipelines. A crucial insight is that while precise interpretation of NL continues to be challenging, restricted versions of natural languages are starting to become practical as natural interfaces in complex decision-making domains. Also, we recognize that DS workflow components are often templated. Putting these two insights together, we develop an initial prototypical system, called Ava, that uses an unambiguous version of NL known as Controlled Natural Language (CNL) to program DS workflows.

The initial Ava system was limited in functionality and focussed on training Machine Learning (ML) models using NL conversations. However, real-world data analysis scenarios go far beyond just training ML Models and involve various stages like exploratory analysis, statistical analysis, and data cleaning. Some analyses may require slicing, dicing, querying, and visualizing the data in different dimensions. To cater to

these broader analytics tasks, we propose modifications to Ava’s CNL and formalize Ava’s CNL via a pattern language. We also propose an end-to-end system architecture for an NL-based data analytics system which is deployed commercially as DataChat.

Finally, we propose ML pipelines, a flexible, scalable, and framework-agnostic approach to compose ML model training workflows using DataChat’s CNL. We also present DataChat’s AutoML framework, which is built using the ML pipeline framework. We evaluate the efficacy of training ML models using DataChat via a user study and find that DataChat’s ML infrastructure improves non-expert users’ productivity by enabling them to compose high-quality ML model training pipelines easily compared to other programming tools such as Python.

1

Introduction

It is now increasingly common for many aspects of business decisions to be guided in some form by data. This change in approach permeates through every level of the organization, from the CXOs down to the business analysts. However, the task of analyzing data resides in a select subset of employees, the so-called DS professionals i.e., data scientists or data analysts. These professionals are responsible for translating complex business problems into a technical form, then applying their data analytics skills to solve these problems, and finally producing insights to improve business outcomes. To solve the business problem, the DS professional starts by collecting relevant data and explores various tools and techniques to extract insights from data, building what is known as a DS workflow.

1.1 The Data Science Workflow

In this section, we describe and define the modern DS workflow. We begin with an example and then formalize key terminology.

Example 1.1. *A data scientist Daisy starts with a business problem, namely: “Predict which customers are most likely to respond to a direct email offer for a personal loan”. As illustrated in Figure 1.1, she begins by loading some relevant*

data about customers from databases or files. She may use basic statistics and visualizations like histograms or scatter plots to understand the data better as well as gain intuition into the relevant features. She may find the need to impute missing postal Zip codes or bucket dates by month. She proceeds to carve out training and test datasets, and build a classification model using logistic regression. Finding that the model has poor precision and recall, she decides to use a decision tree instead. After confirming that the classification accuracy has improved, she fine-tunes various hyper-parameters such as tree depth or splitting criterion. She may iteratively visualize the tree or use a ROC curve to guide this model tuning. Finally, she settles on a model and applies it to current customer data to make her predictions about which customers are most likely to leave/churn.

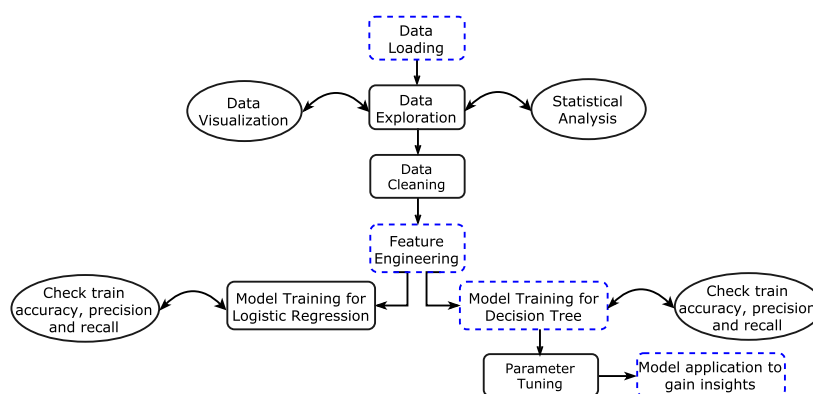


Figure 1.1: The DS workflow for Example 1.1. Tasks are shown in rectangles, and meta-tasks in ovals. The pipeline is highlighted in dotted blue boxes.

As illustrated in the example above, the DS **workflow** is composed of various logical **stages**, such as *Data Loading* and *Model Training*. At each logical stage, Daisy performs a number of physical **tasks**, such as loading one or more datasets from CSV files or databases, or training a *Decision Tree* model. In fact, while Daisy goes through roughly the same sequence of logical stages to model every business problem, she may perform very

different tasks at each stage. For instance, when asked to predict inventory needs for the holiday season, she may go through a workflow with similar stages, but load data from different sources and train a regression model rather than a classification one.

Thus, the data scientist faces a number of choices of tasks at every stage, and must decide which task is best suited for this problem. These decisions are informed by prior experience and rules of thumb (e.g., decision trees might not work well if the number of observations is very small). In addition, these decisions are guided by **meta-tasks** such as statistical analysis and visualization. For instance, histograms and null-value counts are used to guide the imputation method used for data cleaning, and the choice of bucketing or encoding used for feature engineering. Similarly, metrics such as precision and recall (for classification) or R^2 (for regression) and visualizations such as Receiver Operating Characteristic (ROC) curves are used to guide model selection, including hyper-parameter tuning.

The outcome of this exploratory process is a DS **pipeline** that results in a model used to gather insights and solve the business problem at hand. The pipeline (shown in dotted blue boxes in Figure 1.1) excludes the extraneous steps (shown in black) that were incidental to the exploration but will be cleaned up in the final output. For example, when asked to repeat the process for the same problem again, Daisy can use the already cleaned data (foregoing the *Data Cleaning* stage), directly train only the *Decision Tree* (skipping the mis-step of training the *Logistic Regression* model), and may even skip the meta-tasks like data and model visualization. Only this subset of tasks forms the pipeline in our terminology.

A **workflow** is thus a graph (usually cyclic) consisting of **tasks** and **meta-tasks**, conceptually grouped into **stages**. The **pipeline** is a subgraph of the **workflow**. The DS problem is one of **searching** for the best pipeline, and the exploratory steps involved make up the workflow.

1.2 Characteristics of a Data Science Workflow

Data science workflow is templated The DS workflow is composed of a small, predefined set of stages and a well-defined sequence in which they occur in the workflow (possibly iteratively). Each stage is itself composed of several tasks.

```
from sklearn import tree
classifier = tree.DecisionTreeClassifier(
    criterion={{criterion}},
    max_depth={{max_depth}})
model = classifier.fit(training_set, labels)
```

Listing 1.1: Code template for training a decision tree

The tasks in the workflow are typically performed by writing code in a general-purpose programming language like Python, R or Scala, and using libraries like scikit-learn [1], caret [2] or Spark [3]. Usually, the code used to perform a certain task follows a corresponding template: a certain sequence of library calls. Listing 1.1 shows a code template for the task of training a decision tree using the Python scikit-learn library, wherein the data scientist is only required to select and set a few parameters, such as node splitting information criterion `criterion` and decision tree depth `max_depth`. Thus, the semantics of tasks typically lend themselves to a clean separation of specification (parameter values) and template, making it possible to compose the task code simply by substituting parameters into a pre-defined code template.

Data science workflow is exploratory Another critical feature of this workflow is that it is highly exploratory. At every stage of the workflow, the data scientist must choose the most suitable task and parameters from a large number of choices.

1.3 Motivation

A key challenge in DS is the search problem of identifying the best data science pipeline for the business problem at hand. A data scientist should have a solid understanding of the business problem to produce any meaningful insights that the enterprise can directly use.

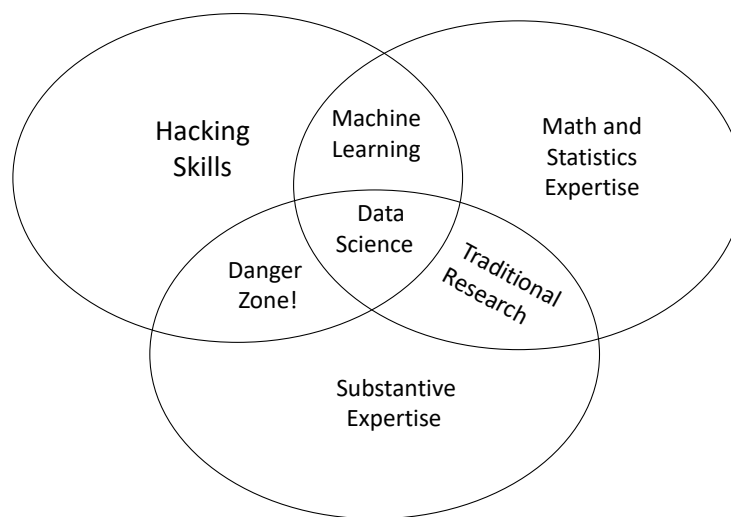


Figure 1.2: The Data Science Venn diagram

Figure 1.2 shows a Venn diagram that describes the skills that are necessary for a data scientist [4]. Note that substantive expertise is one of the skills. A data scientist must understand the business requirements to ask the right questions and come up with the relevant insights. Unfortunately, many data science projects in organizations today do not provide the intended insights due to the impedance mismatch on the part of the data scientist in translating the business requirements to a DS workflow [5, 6]. The continually evolving DS ecosystem has provided data scientists with a plethora of programming languages, libraries, and tools that have greatly

improved the productivity of the data scientists in building pipelines and workflows. Despite the remarkable advancements, the data scientist is still left with the time-consuming steps of learning how to use the various tools (each with its pros and cons) and gluing together tasks performed using these tools into a workflow, all while faced with the challenge of making decisions to navigate the search space of possible pipelines. Thus, the diversity of available tools, while a boon in many ways, has the detrimental effect of increasing the cognitive burden on data scientists and steepens their learning curve.

While we should continue to improve these tools, we must also be mindful of the broader organizational context for DS workflows. The real bottleneck is the turn-around time for the *entire* workflow. A crucial aspect of this holistic view is to improve the human productivity of the data scientists and reduce their turn-around time in assembling DS pipelines. We note that our broader vision is to enhance the productivity of the entire enterprise data-to-insights workflow and democratize DS capabilities throughout an organization. In this thesis, we start with the data scientists, as it provides a concrete starting point. In the future, we plan to target increasingly broader user-bases consisting of ever-less technical users, empowering them to construct and manage robust DS workflows directly.

1.4 Approach

To improve the productivity of data scientists, we begin with the observation that a NL interface improves the efficiency of translating business problems into the technical domain, by reducing the impedance mismatch. We note that various tools available in the data science ecosystem overlap in the problems they solve. We propose using an NL interface to *abstract* away the details of the underlying libraries and let a user cleanly specify their task at the conceptual level, automating the translation to executable

code. Specifically, we use a restricted form of NL called Controlled Natural Language (CNL) to make the NL conversations precise and avoid ambiguity.

The second key observation is that the templated nature of the workflow facilitates this translation to executable code. Each task in the workflow (e.g., “Train a decision tree model with the given set of hyper-parameters.”) maps to a short code sequence specific to the underlying library or platform in use. By maintaining a library of such task templates, we can translate a task specification in natural language into executable code by instantiating the template.

Further, we note that the DS workflow often proceeds through a well-defined sequence of logical stages, from data loading and cleaning, through feature engineering, to model training and visualization. This observation prompts us to build workflow tools that guide the data scientist through these stages by *composing* the templated tasks at each stage.

1.5 Contributions

This thesis makes an initial attempt to improve the human-efficiency in building data science workflows by proposing to assemble data science workflows using NL (CNL). The main contributions of this thesis are:

1. the design and implementation of a CNL engine
2. the end-to-end design and architecture of a system that allows data scientists to complete a variety of data science tasks such as data exploration, data visualization, data querying, and machine learning using CNL.

This thesis consists of four chapters, in which we present the initial end-to-end architecture for an NL -based data analytics system that uses CNL to represent user intent. We formalize CNL via a pattern-based language and

explore the use of CNL to build Machine Learning (ML) model training workflows via a user study.

Chapter 2 We begin by introducing Ava, a DS conversational agent that uses CNL to help data scientists compose workflows. We present an initial system design and architecture for such a conversational agent. We also conduct user studies and show that the agent can improve the productivity of data scientists.

Chapter 3 We formalize Ava’s CNL specification as a pattern based language with entities and intents as the building blocks. We also propose an end-to-end architecture for an NL-based data analysis system. We finally present the lessons learned from commercially deploying Ava as a system called DataChat.

Chapter 4 We present the design of DataChat’s ML pipeline, a Directed Acyclic Graph (DAG) based framework that enables the construction of scalable, reusable, framework-agnostic machine learning ML workflows. We also describe DataChat’s AutoML framework, which uses the ML pipeline infrastructure to automate the search for high-quality ML models. We evaluate the efficacy of using DataChat’s CNL to compose ML model pipelines via a user study and present our results.

1.6 Comparison with other NL-to-task mapping approaches

There has been extensive research on translating NL specifications of tasks to domain-specific task representations [7]. In the context of data analytics, systems such as NaLIR [8], Athena [9], and Eviza [10] NL is translated to specific data analytics tasks such as data querying via SQL and data visualization.

In these systems, the translation task entails mapping task specifications in a single task domain (NL) to a single target domain (SQL). In

contrast, we seek to generate translations of NL task specifications that span multiple task domains such as data querying, data cleaning, data visualization, and machine learning by composing task mappings to target domains such as Python, R, Javascript, and SQL.

For data science tasks in which the task specifications span multiple domains, generating translations is a two-step process that involves:

1. identifying the task domain, e.g., data querying, data visualization
2. mapping from the NL task specification to a target task representation

Most NL-to-task translation systems use learning-based approaches to perform the NL-to-task mapping. These learning approaches require the availability of high-quality NL task-to-target domain training example pairs for every task specification that is to be learned by the model. For data science tasks, there could be several such task specifications that the model needs to learn. For example, just for data visualization, the model should be trained to differentiate between different plotting tasks, such as drawing violin charts, bubble charts, and scatter plots with subplots.

Such models that are trained to learn several classes are susceptible to distribution shifts in the training data. During our initial experiments using ML models to classify data science task domains from NL sentences, we found that the mappings generated by the model were probabilistic. Re-training a model with additional samples to avoid misclassifications for a particular class caused shifts in the decision boundary of the model, which resulted in misclassifications that we did not observe with our initial model.

Additionally, these ML models are inherently error-prone as they seek to translate directly from ambiguous and imprecise uncontrolled natural language to the target task domain.

The sensitivity of ML models to distributional shifts [11] and frequent misclassifications makes using these models intractable for mission-critical applications such as data analysis in enterprises.

Our approach aims to generate deterministic task mappings from NL sentences to various data science tasks. We achieve this deterministic task mapping by restricting the space of task specifications to a restricted form of NL called Controlled Natural Language (CNL) that can be unambiguously parsed using information extraction techniques.

1.7 Future Work

The democratization of data requires dramatic simplification of the tools necessary for data analysis. This thesis proposes a sui generis NL interface for performing data analysis tasks. Feedback from the user studies and learnings from deploying the system in industrial settings have opened up exciting avenues of future work including, but not limited to:

- Model management for managing and reusing ML pipelines across various modeling tasks.
- Enabling human-in-the-loop analytics by facilitating human-to-human communication and collaborative data analysis.
- Developing multi-modal interfaces for complex analytics tasks.
- Making NL analytics accessible to a broader set of audiences by relaxing DataChat’s CNL specification.

1.8 Scope and Limitations

End-to-end data analysis workflows are complex and involve various stages such as Exploratory Data Analysis (EDA), data cleaning, statistical

analysis, to name a few. This thesis focuses on the ML component of the data science workflow while providing a broader systems perspective and architecture for building an end-to-end NL data analytics systems.

2

Ava: An Intelligent Conversational Agent for Data Analysis

We introduce Ava, an intelligent conversational agent that is aimed at simplifying process of constructing data science workflows and pipelines. It uses a natural language chat-based interface to abstract away the details of the underlying ML and data analysis/visualization environments, letting the user focus on the business problem.

In this chapter we present the system design and architecture of Ava. We also conduct user studies and show the efficacy of Ava when Ava is employed to solve DS tasks.

2.1 Related Work

There has been significant research in understanding natural language, dating back to the “Imitation Game” proposed by Turing in 1950 [12]. The quest to demonstrate an Artificial Intelligence (AI) capable of behaving indistinguishably like humans spurred research into early chatbots such as ELIZA [13], PARRY [14] and ALICE [15]. The Loebner Prize competition [16], conducted annually since 1991, was the first formal instantiation of a Turing test.

In recent years, the explosion of mobile web usage as well as improvements in natural language parsing have led to a resurgence in chatbots. These chatbots are designed with much more pragmatic ends in mind, such as virtual assistants that help book flight tickets or order pizza delivery. Many recent language understanding services such as Microsoft LUIS [17], Watson Conversation [18] and Amazon Lex [19] are designed to ease development of such chatbots. Due to their closed-source nature and licensing restrictions, however, we chose to build our research prototype Ava using the open-source AIML [20] chatbot framework.

An important observation is that practical conversational interfaces can function reasonably well while understanding only a restricted subset of natural language. Such a “controlled natural language” (CNL) removes ambiguity in interpretation, thereby easing development of such interfaces. This approach has therefore found increasing use, including in air traffic control [21, 22]. Some CNLs can also be mapped to first-order logic for unambiguous interpretation (e.g. [23, 24]), and we build on this insight to control Ava’s chat language.

There has been a large body of research aimed at simplifying data access for non-programmers through the use of natural language, dating back to late 1960s. We direct the interested reader to the comprehensive survey [25] of NL interfaces for databases (NLIDBs) to review the progress made over several decades. In the early NLIDBs, the acceptable input query grammar was usually tailored to the specific data schema, reducing their general applicability. More recent work, including Microsoft English [26], Precise [27], Nalix [28], Nalir [29], Athena [9] and Nauda [30], has sought to overcome this shortcoming by translating to SQL in a generic fashion. While systems such as Microsoft English [26] and Athena [9] rely on a semantic model of the database to correctly interpret natural language queries, others like Precise [27] and Nalir [29] use only the names of database elements and their synonyms. Most recently,

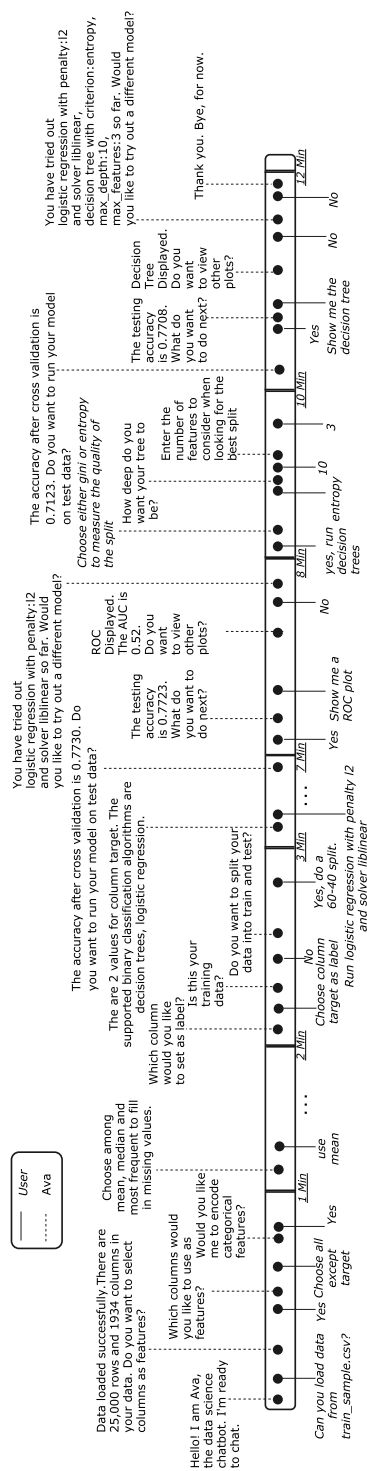
EchoQuery [31] demonstrated a conversational, speech-based interface. Ava too advocates using natural language as the interface to data systems. We believe research on NLIDBs to be complementary to our own, since we target the broader data science domain rather than the database interface.

There is broad recognition that various aspects of DS pipelines must be optimized (e.g. [32–34]), and we have a similar overall goal. We advocate a specific approach for part of this problem using chatbots; many other aspects in previous work, such as smarter parameter selection, are complementary with our proposed approach.

2.2 System Design and Architecture

Ava interactively guides the data scientist through the process of deciding among the various tasks to be performed at every stage of the workflow. It does so using a conversation storyboard that mimics the decision-making process of a data scientist.

Example 2.1. *We have used Ava to execute the workflow in Example 1.1 in an initial user study. Figure 2.1 shows a timeline view of the resulting conversation for a sample user. Ava interactively guides Daisy through the successive stages of the workflow. At each stage, Ava uses natural language questions to clarify Daisy’s intent and then executes the appropriately instantiated code template. Ava also proactively reports meta-information that might help Daisy take the next action, like automatically reporting the cross-validation accuracy after training a model. In future, Ava will also learn to make recommendations, such as proposing that a decision tree may be a more accurate model than logistic regression in a similar context.*

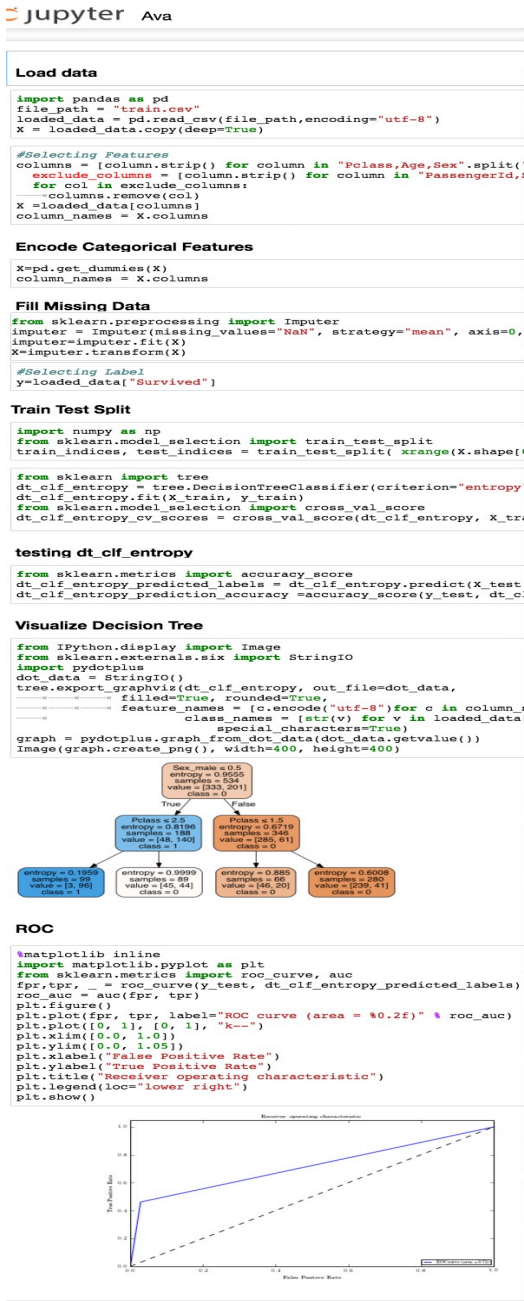


2.2.1 Client-Server Jupyter Notebook Design

The conversational nature of Ava naturally lends itself to using a notebook-style of interaction. Ava currently uses the Jupyter Notebook environment. This notebook style also allows a data scientist to “take over” from Ava and switch to their current mode of operations. The Jupyter Notebook [35] is an interactive computing environment that is widely used by data scientists. As the user types in code (typically in Python or R), the code can be executed, and results, including tables and figures, are produced inline with the interface. Thus, a notebook can be produced with the code, intermediate results, and the final results. Jupyter also has a client-server architecture. In the server side of the Jupyter Notebook framework, there is a programming language-specific kernel. This kernel manages the interactive context and controls code executions that are triggered by the client. The client communicates with the kernel using a RESTful API. It is possible to run the client and the server on different machines. Ava taps into this existing architecture of Jupyter as described in more detail in Section 2.2.2.

2.2.2 Control Flow Architecture

At a high-level, Ava is built on the Jupyter Notebook framework. An extended IPython kernel serves as the chat server. The chat server is responsible for parsing the chat messages, extracting the semantic information about the task to be performed, identifying when further clarification is required, and finally generating the executable code. The Ava chat client has an instant messaging-like interface, and runs above the regular notebook client, as shown in the screenshot in Figure 2.2. The user interacts directly with the chat client, providing instructions and answering questions using chat messages. When the specification for some task is completed, the Ava chat server triggers Python code generation.



The notebook code includes the following sections:

- Load data:** Imports pandas and loads 'train.csv' into a DataFrame X.
- Selecting Features:** Uses `drop` to remove 'PassengerId' and `drop_duplicates` to clean the data.
- Encode Categorical Features:** Uses `pd.get_dummies` to convert categorical variables into binary features.
- Fill Missing Data:** Uses `sklearn.impute.SimpleImputer` to fill missing values with the mean.
- Train Test Split:** Uses `sklearn.model_selection.train_test_split` to split the data into training and testing sets.
- testing dt_clf_entropy:** Trains a `DecisionTreeClassifier` with `criterion='entropy'` and evaluates its accuracy on the test set.
- Visualize Decision Tree:** Uses `sklearn.tree.export_graphviz` to generate a decision tree structure, which is then visualized using `pydotplus` and `graphviz`.
- ROC:** Uses `sklearn.metrics.roc_curve` to generate an ROC curve, which is plotted using `matplotlib.pyplot`.

Ava: Your Data Science assistant

You: Survived

Ava: To evaluate your model we need train and test data. Do you wish to load the test dataset? You can also split the data into train and test.

Hint: You can instruct me to split the data by saying "do a train_split_percentage - test_split_percentage split". e.g "do a 60-40 split".

You: do a 60-40 split

Ava: Splitting the data into train and test ...

Ava: There are 2 values for the column Survived. The supported binary classification algorithms are decision trees, logistic regression and SVM

Hint: Type "Run name-of-the-algorithm" to select an algorithm. e.g. "run logistic regression"

You: run decision trees

Ava: Pick a splitting criterion for the decision tree. The choices are gini and entropy.

You: entropy

Ava: How deep do you want your decision tree to be?

Hint: Enter an integer value. Deeper decision trees are more complex.

You: 2

Ava: Enter the number of features to consider when looking for the best split. The choices you have are:

- An integer number, which will set the `max_features` at each split
- A decimal number, in this case, `max_features` is a percentage and `int(max_features * n_features)` features are considered at each split
- If auto, then `max_features=sqrt(n_features)`
- If None, then `max_features=number of features`

You: auto

Ava: The accuracy after cross validation is 0.785. Do you want to run your model on the test data?

Hint: Type "Yes" to see the accuracy of the algorithm on the test data.

You: yes

Ava: The testing accuracy is 0.776. What do you want to do next?

Hint: Type "show ROC plot" to see a ROC plot of the classifier you chose. If your current choice of classifier is decision trees type "show decision tree" to visualize the tree.

You: show decision tree

Ava: Decision tree generated. Do you want to view any other plots?

You: show roc

Ava: ROC generated. Do you want to view any other plots?

Hint: Type "show ROC plot" to see a ROC plot of the classifier you chose. If your current choice of classifier is decision trees type "show the decision tree" to visualize the decision tree. Type "No" to move on to the next conversation.

You: No

Ava: You have tried

- decision tree with `max_features:auto`, `max_depth:2`, `criterion:entropy`

so far. Would you like to try out a different model?

Hint: Type "yes" to select a new algorithm. Type "No" to exit the conversation.

No

Figure 2.2: Ava in action. The right panel is the chat window. The conversation between the user and Ava is carried out in this chat window. Actual code is created and run in the left panel using the regular Jupyter Notebook interface.

After executing this code, Ava updates the Jupyter Notebook with both the code and its result. Thus, a user can take over from Ava at any point and directly write code in the notebook. This aspect is crucial, as it allows advanced users to jump back to the usual mode of “driving” the data analysis process, minimizing frustration when the task at hand can’t be done using Ava. (Although, at this point, the one-to-one correspondence between a chat and the associated code is lost.)

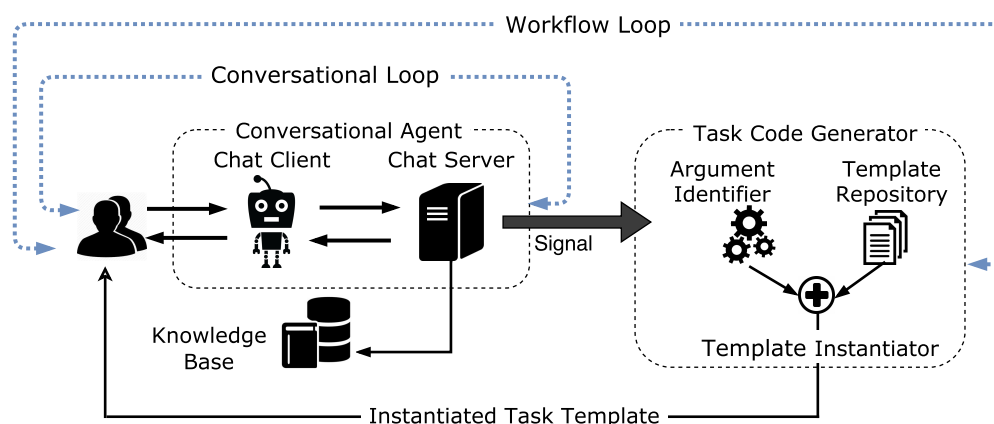


Figure 2.3: Overview of Ava’s control flow architecture.

Figure 2.3 shows Ava’s control flow architecture. The user chats with the *conversational agent* in a Controlled Natural Language. The conversational agent consists of a *chat client* and a *chat server*, and matches the input with patterns in the AIML files (more details about how Ava uses AIML is presented below in Section 2.2.4). The conversational agent is responsible for steering the conversation towards a full task specification.

The chat server consists of a pair of processes, one that is the AIML interpreter (a Java process), and the other that is the iPython kernel. The chat client sends the messages that are typed by the user to the iPython kernel, using iPython’s REST-based communication API.

The IPython kernel then passes the chat message that it received to the AIML interpreter (which is part of the chat server). The IPython kernel communicates with the AIML interpreter using JPype [36]. Once the AIML interpreter matches the chat message with a pattern in the AIML files, it sends a response back to the IPython kernel using JPype. The IPython kernel, then relays back the response to the Ava chat client using the kernel's REST-based API. The conversation between Ava and the user continues until a full task specification is reached.

Once a specification is complete, the system signals the *task code generator*, which looks up its *template repository* to identify the template that best matches the task specification. Its *argument identifier* extracts the template parameters from the chat messages, and the *template instantiator* fills out the chosen template with the user-specified parameter values. The generated code is now executed on the underlying DS platform. Both the code and the results are also presented back to the user in the Jupyter Notebook client. The chat server logs all interactions to be compiled into a *knowledge base* and consults it to guide the workflow towards the optimal pipeline.

Note that the discussion above described how the template code generates Python code to run in the Jupyter Notebook client. This same mechanism can be used to call out to other ML libraries by simply having the template code invoke APIs in the ML platform (e.g., TensorFlow), which can be run as a separate process. Ava is currently packaged as a Docker [37] image, which also allows for easy packaging and deployment of all the associated components.

Further, Ava automatically suggests tasks as well as presents statistical data and visualizations to aid the data scientist in the process of exploratory analysis. We envision that Ava will eventually manage data movement between tasks (orchestrating multiple subsystems such as databases and ML libraries) as well as between machines (intelligently spawning appropriate instances in the cloud on-demand), thereby resulting in further

improvement in productivity.

However, we acknowledge that the capabilities of such an interface will likely lag behind the fast-evolving libraries in the underlying ecosystem. To minimize any frustration on the part of the experienced data scientist, Ava *always* allows a user to seamlessly drop into a familiar programming interface (the Jupyter Notebook) at any stage in the workflow.

Throughout this process, Ava logs its interactions with the data scientist, producing a rich dataset that we will eventually compile into a growing knowledge base for DS pipelines. We envision that this process of constantly learning from user interactions will make Ava more intelligent over time, helping data scientists find the optimal pipeline ever faster.

We note that the chat logs provide a natural way to document the choices that are made. Chat logs are easier to read (compared to examining a stitched sequence of code), to review/revisit decisions, and to reuse portions of the decisions in building future pipelines.

2.2.3 Controlled Natural Language (CNL)

While NL is highly expressive and capable of representing arbitrary concepts, this expressiveness comes with ambiguity and poses a challenge for automatic semantic inference. CNLs [23] provide a middle ground that balances the expressiveness of natural language with ease of semantic inference by restricting the vocabulary and grammar in the language. Thus a CNL constrains the input to be more concise, with a less general sentence structure, making it easier to parse and extract meaning from the input sentences. (CNLs have been used successfully in many domains, including air traffic control [22].)

The sentence structure for Ava’s CNL is: <ACTION VERB> <NOUN> <PREDICATE>. For example, a data scientist can say *Run logistic regression with penalty l2 and solver liblinear* to instruct Ava to fit a logistic regression classifier. Here, “Run” is the ACTION VERB, “logistic regression” is the NOUN and

“with penalty 12 and solver liblinear” is the PREDICATE. This simple sentence structure makes it easy for Ava to identify the action to be taken in response to the user’s input as well as the parameters associated with the action.

2.2.4 Artificial Intelligence Markup Language

In addition to CNL processing, Ava has to maintain a conversational context. We briefly describe our approach here.

The conversational context is represented by a state machine, which has the following key elements:

- The current state of the conversation, e.g., *What stage of the workflow are we in? What are the dataset and model variables in scope?*
- The allowable inputs that will trigger a state transition, e.g., *What can the user say at this stage in the workflow to choose a task or move to the next stage?*
- The corresponding actions that Ava must take when the user effects a state transition, e.g., *What code should be executed when the user asks to train a decision tree?*

The state machine is wrapped up with a layer that implements the actual chat/conversation. We use the popular XML dialect, Artificial Intelligence Markup Language (AIML) [20], to represent the state machine for our chatbot. Each unit of conversation in AIML, called a category, consists of a pattern that represents a possible user input and a template that represents an action that the chatbot can take. An example listing is shown below:

```
1 <aiml>
2   <category>
3     <pattern>HELLO</pattern>
```

```
4 <template>Hi, there !</template>  
5 </category>  
6 </aiml>
```

Listing 2.1: A Sample AIML pattern pattern

We have created various AIML files to script each possible interaction that Ava can have. We also use existing open-source libraries for parsing AIML and running a conversational agent, using developer-defined callbacks for processing user input and effecting state transitions.

2.2.5 Task Template Repository

As we observed in Chapter 1, the code for executing a DS workflow task is templated. In Ava, each such template is represented using a text file with placeholders for the various parameters that can be set. We maintain a repository of such templates, along with a mapping from specific tasks, such as *Model Training for Decision Tree*, to the corresponding code template file. These templates are specific to the underlying libraries or tools that Ava uses and are manually created by us in this initial implementation. In the future, we plan to augment it using static analysis techniques to automatically generate templates from the source code and documentation of the underlying library.

2.2.6 Conversation Storyboard

Ava's conversations with the user are guided through a *storyboard*, which is represented as a set of AIML files. The storyboard describes the dialogue between the data scientist and Ava in CNL, and Ava's actions in response. These responses are often questions that Ava must ask the user to disambiguate their intent and arrive at a complete task specification.

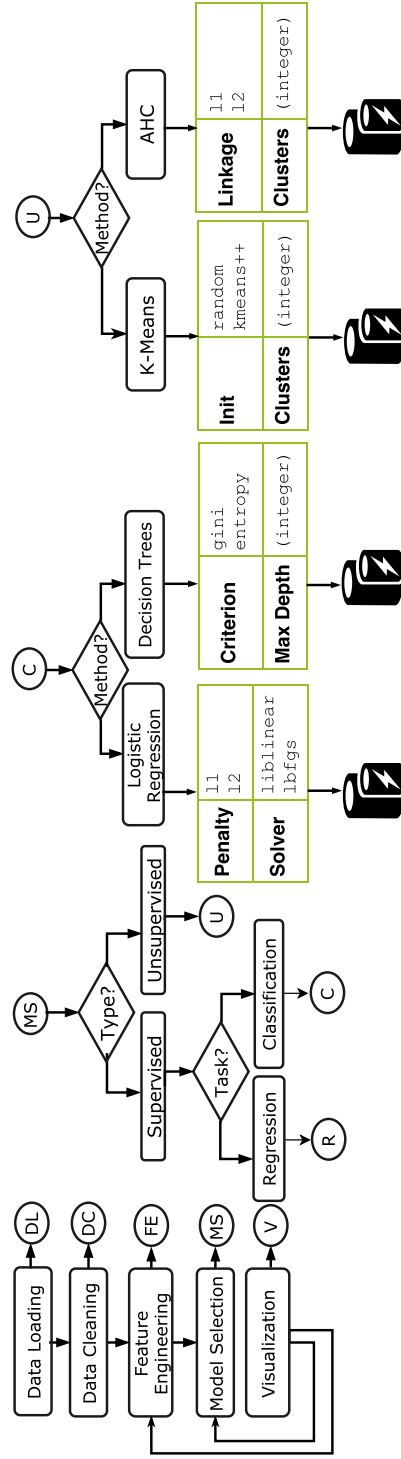


Figure 2.4: The storyboard powering Ava, excerpted and truncated for brevity.

Figure 2.4 visualizes an excerpt of the storyboard in the form of a set of flowcharts. The first flowchart is a high-level view of the different stages of the DS workflow. Corresponding to each stage, the figure shows a separate sub-flowchart. The *Model Selection (MS)* flowchart describes how Ava uses multiple prompts to identify whether the user is interested in a classification, regression or clustering method. The flowcharts for *Classification (C)* and *Unsupervised Learning (U)* further detail how Ava talks to the user to identify the specific method (say *Decision Trees*) and its parameters (information criterion and maximum tree depth). When the task specification is complete, the storyboard also instructs Ava to load the corresponding task template and execute it.

Therefore, a particular workflow plays out as a conversation with Ava, which is modeled as a walk through the flowchart graph. Note that the storyboard is also a template for the entire DS workflow (in contrast to the task-specific template library described in Section 2.2.5). Over time, as Ava grows its knowledge base, we envision that much of the storyboard will be automated, reducing the need for user prompts. For instance, Ava may eventually learn what decision tree hyper-parameters to use so that those prompts in the storyboard are replaced by lookups in the knowledge base.

2.2.7 Salient Aspects

The finite state machine (FSM) framework is crucial to Ava’s ability to unambiguously convert from conversations in CNL to executable code. The FSM states, through the use of CNL, make interpretation of the conversation tractable and unambiguous. The FSM transitions allow Ava to drive the conversation towards a complete task specification. Finally, the task templates allow Ava to generate and execute the code for a task, decoupling underlying libraries from the chat engine.

Ava’s modular architecture allows implementation of each component

using the appropriate technologies. This modularity also allows us to easily extend Ava’s capabilities, such as adding new ML, methods, or libraries.

2.3 Experiments

We designed single-subject and multiple-subject trials to analyze the efficiency of Ava in constructing DS pipelines. We present our experimental design and results in this section.

2.3.1 Single-Subject Trial

We conducted this experiment with a single user playing the role of Daisy in the DS workflow in Example 1.1. We chose the specific Kaggle task [38] where the goal is to identify customers for direct marketing. For training, we used a 15% sample of the 1GB customer dataset, consisting of 25000 rows and 1934 columns. We ran this experiment on a MacBook with a dual-core Intel i7 processor and 8GB of memory. We configured Ava to generate code using the Pandas, Matplotlib, and scikit-learn libraries. Our current Ava prototype has a small number of hand-written templates in its template repository. The knowledge base consists of a set of hand-written rules providing user assistance (including recommendations) based on prefix-based exact matching of the state of the current conversation.

Figure 2.1 visualizes the conversation flow in the form of a timeline with timings measured from our user study for an actual user. Figure 2.2 shows a screenshot of the interaction with Ava. All the interaction between Ava and the user occurs through chat messages in (controlled) natural language, although Ava also presents the generated code and graphs in the Jupyter Notebook interface alongside the conversation window.

A key objective in this experiment was to study Ava’s *response time*, which we define as the time it takes for Ava to respond to a given user

request, not including any time spent in executing generated code for a task. Our measurements showed that the average response time was 93 ms, whereas the maximum was 558 ms. Thus, we believe that our CNL/AIML framework in Ava has fast response times, which is crucial for a chatbot framework. We note that the total time to assemble the final DS pipeline, in this case, was 12 minutes (this includes the time to run the machine learning models in scikit-learn). Of these 12 minutes, only 2 seconds were spent in Ava’s client-server engine. We expect novice users with this DS problem to spend many more minutes putting together the same pipeline by writing code. Additionally, since an expert user can always drop down to the notebook programming interface, Ava does not significantly impair their productivity even in the worst case (2 seconds in this case). We believe that Ava’s proactive presentation of meta-information and recommendations would likely improve productivity for even the experts.

2.3.2 Multiple-Subject Trials

Inspired by the results of the single-subject trial, we conducted a more detailed study to test our hypothesis that Ava significantly improves productivity of its non-expert users, with only a small learning curve. In this study, we asked users to complete the same data science task using both Ava and Python packages and measured various metrics of productivity.

Participants. We recruited graduate students from the Computer Sciences Department at the University of Wisconsin-Madison for this experiment. All students in this user group had exposure to machine learning and data science through their coursework but were not otherwise experienced in these domains. We administered a Python proficiency test and picked 16 participants who passed this test.

Task and data description. The task that we chose for this study was one of predicting customer churn for a telecommunications firm [39]. The

objective of the task is to develop a model to classify which customers are most likely to refrain from renewing their contract when it ends. We obtained a dataset consisting of historical details and account usage information for customers of the telecommunications firm, along with labels that indicate whether or not they renewed their contract. These labels, however, were skewed in that only 433 of the 3333 customers had terminated their contract. We sampled a subset consisting of 2300 customer records and provided them as training data to the participants. The remaining 1033 customer records were used as a held-out test set to evaluate the models developed by the participants. All participants used the same machine, which was a MacBook with 2.8 GHz Intel Core i7 processor and 16GB of memory.

Each participant performed this task once using Ava and once using the Python packages (pandas and scikit-learn). In both cases, the participants were required to follow a 10-step procedure to develop a DS pipeline. The pipeline included steps for feature selection, model training, cross-validation, and visualizing data that described the accuracy of the constructed model(s). For simplicity, we limited the choice of machine learning algorithms to logistic regression, support vector machines and decision trees. We limited the duration of the task to 30 minutes when using Ava and to 60 minutes when using Python. Participants were instructed to explore as many models (combination of learning algorithm, features and hyper-parameter choices) as they could within this duration. They were free to use any documentation, tutorials or code samples from the Internet to help them complete the task. Note that the participants were given no training or instruction on the usage of Ava.

More details about the dataset as well as the exact task description are available at [40, 41].

2.4 Results

Productivity Improvement. We used three different metrics to measure productivity. These metrics are: a) the time taken to complete the first model, b) the number of models that were constructed in a fixed amount of time, and c) the time taken to complete successive models after the first model was constructed. As an overview of our findings (described in detail below), we saw big productivity improvements in each of these three metrics with Ava.

Note that all time measurements include the actual processing time for the analytics steps (within the underlying ML and the visualization libraries), in addition to the time spent by participants in manually proceeding through the steps in the analysis task.

First, we consider the time taken by the participants to train their first working ML model. The distribution of the results using this metric is shown as box plots [42] in Figure 2.5.

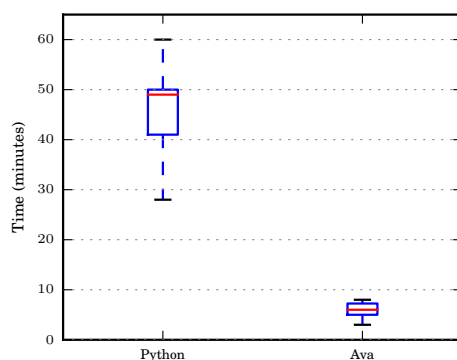


Figure 2.5: Distribution of the time taken by participants to complete the first model.

Of the 16 participants, only 13 completed their first model using Python within the allotted task duration of 60 minutes. On average, these participants took 45 minutes to do so. On the other hand, all 16 participants completed their first model using Ava within 14 minutes, with an average

time of about 7 minutes. Thus, we observed an improvement of more than $6\times$ in this productivity metric.

Second, we consider the number of models that the participants could train within the allotted duration using each approach. Only one of the participants was able to train more than one model using the Python packages within 60 minutes. On the other hand, every participant could explore at least 7 models using Ava, and on average, each participant explored about 11 models within the allotted 30 minutes. Thus, Ava produced at least an order of magnitude improvement in productivity by this metric.

Finally, we note that the participants spent less than 3 minutes training each subsequent model after the first one using Ava. This time was about half a minute faster than the average time per successive model using Python for the single participant that completed training of more than one model.

Learning Curve. Of the 16 participants in the study, half were randomly chosen to perform the task using Python packages before doing so with Ava (we denote this group using the label *Group-PA*). The remaining half were asked to complete the task using Ava before doing so with Python (indicated as *Group-AP*). Comparing the productivity differences between these groups using the two approaches allows us to reason about the learning curves with these two approaches. The distributions of the time taken to train the first model for these two groups are shown in Figure 2.6.

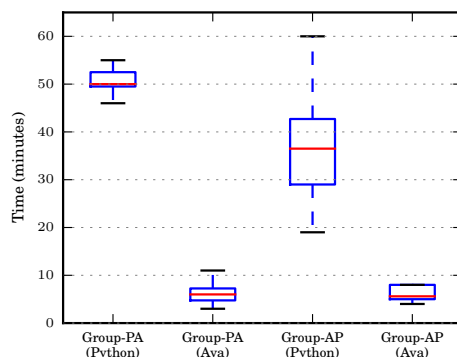


Figure 2.6: Distribution of the time taken by participants to complete the first model in each user group.

When performing the task using Python packages, the 8 participants in Group-PA (who used Python before Ava) took, on average, 51 minutes to train their first model. On the other hand, the 8 participants in Group-AP (who used Ava before Python) took 7 minutes to train their first model.

Figure 2.7 compares the time taken by the participants to train the first working model in Python and Ava when they did not have any prior knowledge of the task.

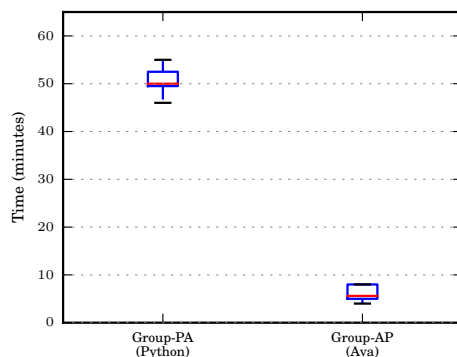


Figure 2.7: Distribution of unbiased task completion times.

Also, familiarity with the task as well as the data science process (by completing the task in Ava first) allowed the participants in Group-AP to take only 27 minutes to reach the same milestone. Contrast this 14 minute

(27%) improvement in productivity with the negligibly small 1 minute improvement (from 7 minutes down to 6 minutes) to train the first model using Ava in Groups-AP and -PA, respectively. Further, we saw only a negligibly small difference in the average number of models trained by the participants using Ava in the two groups.

We consider these observations as indicating that novice users can be nearly equally productive with Ava regardless of their prior experience in or familiarity with the data science process. We cannot, of course, extend these conclusions directly to expert data scientists since they were not part of this study.

Other Observations. Since we held out a subset of the labeled data from the participants in the study, we were able to compare the accuracy of their models on this (unseen) test set. A surprising observation was that the “best” model trained by participants using their cross-validation accuracy scores often had far lower accuracy scores on the test set. Clearly, empowering users with the ability to explore a large number of models in a short time had the unintended consequence of encouraging them to create highly over-fitted models. Furthermore, nearly all the participants seemed to have overlooked the skew in the labels (class frequencies) in the datasets, thereby obtaining models that had much higher prediction accuracies for one class than for the other.

These and other observations indicate that, despite data science training and some hand-holding, users can still make mistakes that impair the accuracy and usability of the DS pipelines they create. A DS workflow tool such as Ava must be built with sanity checks and knowledge of best practices to guide non-expert users away from such mistakes. In particular, we are looking to add checks against over-fitting as well as detection of class frequency imbalance into our Ava storyboard to address the specific issues that we observed. We note that this aspect of providing expert guidance or cautionary flags is an additional interesting aspect of Ava.

Analogous to how automation helps drivers of cars with autonomous features, with Ava, we can build safeguards that warn its user when the user starts to tread into “unsafe” territory.

2.5 Conclusions

The true democratization of data requires dramatic simplification of the methods used to extract insights from data. In Ava, we present a small step in this direction, a conversational agent that targets improving the productivity of data scientists. Ava uses controlled natural language as the “programming” interface and has an architecture for building a knowledge base that can improve the overall productivity in performing data analysis. This initial approach opens the door to numerous open research challenges, including managing and mining the knowledge base, recommending visualizations, automatically building ML models and even larger issues in making the Ava approach accessible to non-data scientists.

3

DataChat: Ava Redesigned

Building off of Ava’s prototypical system design, we’ve developed DataChat. This intelligent system helps users to analyze, to explore, and to derive insights from data using conversations in CNL. We’ve rearchitected Ava to make the system more user-friendly, resource-effective, and scale to handle larger datasets (as often the human user is in the “computational loop”). Further, we’ve improved upon Ava’s CNL specification and implementation to be more robust, flexible, and extensible. This chapter presents modifications to Ava’s CNL and describes improvements to Ava’s system design and architecture.

3.1 Flexible and Robust CNL

As we note in Chapter 2, forcing the user to converse in CNL significantly reduces the ambiguity inherent in uncontrolled NL. However, CNLs that are too restrictive may still have a high-cognitive learning overhead. The CNL used in Ava was inflexible in that the CNL was sensitive to the presence or absence of stop words. The specification did not allow for synonyms or mild variations to the sentence structure. For example, given the Ava CNL specification `Load data from the file <filename>`, Ava would fail to recognize the utterance `Read file titanic.csv`.

To account for such deficiencies in Ava’s CNL, we designed a flexible yet simple CNL by incorporating synonyms and allowing minor variations in sentence structure. It’s important to note that DataChat’s CNL specification, though more expressive than Ava’s CNL, remains a restricted and unambiguous subset of uncontrolled NL. This design of using CNL is still essential, as the key consideration for both Ava and DataChat is to be as precise as traditional imperatively-programmed data science workflows since the *accuracy* of the overall pipeline is critical in enterprise settings.

In this section, we describe the design of DataChat’s CNL.

3.1.1 Terminology

In DataChat, the user interacts with the system through **conversations** that are either driven by the user or by the system. These conversations are imperative sentences. The system presents questions in the form of **prompts** to the user, and the user responds to the prompts using **utterances**.

The system may have a set of computational **tasks** that it is capable of executing. These tasks are exposed to the user through CNL. We call the CNL specification of the tasks as **intents**. Intents can be refined further and made more precise through the specification of values for the **entities**.

3.1.2 Pattern Language

DataChat’s CNL allows for the specification of user intents and entities through a simple pattern language. In the pattern language, we specify a pattern for every intent. Listing 3.1 shows a pattern specification in DataChat’s pattern language corresponding to the task of loading data from a file.

```
1 "LoadDataFromFile" : Pattern(  
2 'LOAD data from the FILE <filepath>',  
3 synonyms={
```

```

4 'load': ['read', 'import'],
5 },
6 addnl_ignore=['data', 'dataset'],
7 entities={
8 'filepath': PositionalEntity(FilePath(),
9 "Where is the source file located?"),
10 },
11 )

```

Listing 3.1: The LoadDataFromFile pattern

Listing 4.1 shows the pattern corresponding to the task of training a decision tree classifier with a particular depth parameter and a specific criterion (either GINI or entropy).

```

1 "TrainDecisionTreeClassifier": Pattern('TRAIN a
    DECISION TREE CLASSIFIER with <criterion> <
    max_features> <max_depth>',
2 synonyms={
3 'train': ['build'],
4 },
5 addnl_ignore=['model', 'and'],
6 entities={
7 'criterion': KeywordEntity(OneOf(['gini', 'entropy',
    'all'])),
8 "What splitting criterion would you like to use in
    building the decision tree? The supported values
    are gini and entropy. Say 'all' to try both of
    the values",
9 ['criterion'])),
10 'max_features': KeywordEntity(Number(),
11 "What is the maximum number of features that you'd
    like me to consider for each split?",

```

```

12 ['max features', 'maximum number of features',
13 'maximum features']),
14 'max_depth': KeywordEntity(Number(),
15 "What is the maximum depth of the decision tree you
    want me to build?",
16 ['max depth', 'maximum depth']),
17 },)

```

Listing 3.2: The TrainDecisionTreeClassifier pattern

The following properties hold for each pattern in DataChat:

- Every pattern specifies a **pattern statement**.
- Conversations in DataChat are in imperative speech as each pattern **begins with a verb**.
- The pattern consists of one or more words known as **pattern identifiers**. The leading verb is one such pattern identifier. Pattern identifiers are used by DataChat's CNL engine (Section 3.2) to match a sentence with a pattern. Every pattern should consist of a non-empty set of distinct identifiers. The pattern LoadDataFromFile has a single identifier load. Each identifier may optionally have a set of **synonyms**. The presence of a synonym in an utterance is considered equivalent to the presence of the corresponding identifier itself. For instance, the LoadDataFromFile pattern has synonyms read and import. An utterance that is matched to a pattern should contain all pattern identifiers or their synonyms.
- Each pattern can have one or more associated **entities**. These entities should appear in the pattern statement as well. The LoadDataFromFile pattern has a single entity called filename, corresponding to the name of the file from which the data is to be loaded. The entity specification indicates the data type of the values accepted by that

entity and the entity's location in the utterance. The entity data types are defined in DataChat's CNL engine.

- Entities can be **positional**, **keyword**, or **prepositional**. Positional entities are identified by the position in the pattern statement and utterance relative to the identifiers, e.g., the filename entity in the LoadDataFromFile pattern. Keyword entities may occur anywhere in the utterance relative to each other: instead of their position – they are parsed based their association (either before or after) with a keyword in the utterance.

For instance, the entity `criterion` in the `TrainDecisionTreeClassifier` pattern in Listing 4.1 can be mentioned as “train a decision tree classifier with criterion gini” or “train a decision tree classifier with gini as the criterion.” Prepositional entities are used to specify a sequence of words that are preceded by a preposition. In Listing 3.3, `<alert>` is a prepositional entity that is preceded by the prepositional phrase with `alert level`. This pattern will match utterances like “predict the future daily usage for the next five days with alert level 1”. Prepositional entities provide a mechanism to specify phrase-level matches instead of just keyword based matches.

```

1  "PredictFutureWaterUsage": Pattern(
2      'PREDICT future daily usage for the next <
   num> days <alert>',
3      synonyms={
4          'predict': ['forecast']
5      },
6      entities={
7          'num': PositionalEntity(
8              Number(), ""),
9          'alert': PrepositionalPhrase(

```

```

10     Number(), "", ['with alert level']
11     ), }, )

```

Listing 3.3: The PredictFutureWaterUsage pattern

- By default, the CNL ignores commonly used stop words and functors [43] in the English language, except for stop words occurring in the prepositional entity specification. Words that carry no semantic information to the pattern can be added to a list of ignored words in the pattern specification. Such words exist merely as syntactic sugar and for the purpose of making the conversation look more natural.

An example of an utterance that matches the LoadDataFromFile pattern is: “Load data from the file titanic.csv”. Words such “data”, “from” and “the” are ignored, and the utterance gets reduced to its essence “load file titanic.csv.” Since “read” and “import” are synonyms for load, the “read file titanic.csv” and “import file titanic.csv” are equivalent utterances.

Essentially, the DataChat CNL has flexibility in matching different sentences with the same intent to a specific task, but this flexibility is controlled so that there is a precise mapping. In contrast, in methods that use uncontrolled natural language mappings to intent approach (e.g. using deep neural networks), it is possible that the same sentence may map to a different intent if the neural network is retrained.

For example, consider the sentence “Find the highest sale recorded in California or Wisconsin”. (This would translate to a SQL task in DataChat.) Two different humans could think of this same sentence in two different ways: one regarding *the* correct answer to start by union-ing both states into one basket and reporting the highest sales across both states, and the other considering *the* correct answer as one that reports the highest sale for each state. (There are other variants of interpretation, especially in the presence of duplicate highest sale values.) One can easily imagine that a trained Neural Network may one day switch from

one interpretation to the other depending on what data gets fed into the retraining method. This un-stability makes it hard to construct reliable deep DS pipelines (essentially writing a “program” as a sequence of NL sentences) in an approach that does not control the grammar “variations” by design. Database aficionados may recognize the similarity to query optimization, where a new “improved” query optimizer that changes the behavior of what used to happen before the improvement wrecks havoc in operations, as the enterprise has to rely on the “old” way continuing to work as they did before. Admittedly, the CNL approach requires human training over an approach that uses (uncontrolled) NL as the DS program “specification” language, but that tradeoff is what we make in our approach as we place a strong emphasis on consistency/reproducibility. Another way to look at our approach is that we have created a context-sensitive grammar specification with natural English syntax.

3.2 CNL Engine

DataChat’s CNL engine is a parser that recognizes NL utterances specified in DataChat’s CNL. When a user submits an utterance such as “load data from the file titanic.csv”, the CNL parser first tokenizes the utterance into words using a Spacy tokenizer [44]. Then, the parser identifies the pattern in the CNL pattern specification corresponding to the utterance. This process is known as intent classification. We note that this intent classification task can be done using various methods ranging from a rule-based method to using an ML-driven NLP method. However, with an ML-based approach, one has to be mindful of the impact of what retraining means (see the discussion above).

In this version of the CNL engine, we’ve implemented the intent classification using a keyword-based approach. Once the pattern (intent) is identified, the engine extracts entities from the utterance. Since Dat-

aChat is primarily designed for data analysis tasks, the entity extractor should recognize complex data types such as dates, timestamps, currency, zipcodes, to name a few.

3.2.1 Intent Classification

The intent classifier (or pattern identifier) is a keyword-based system that matches an utterance with the patterns solely based on the identifiers. The algorithm for intent classification is given in Algorithm 1.

3.2.2 Entity Extraction

Once the intent of an utterance is identified as belonging to a particular pattern, the parser CNL engine extracts entities from the utterance (if any exist). As we describe in Section 3.1, the CNL engine consists of parsers that extract values corresponding to a data type specified by an entity. We begin the entity extraction process by tokenizing the utterances. The tokens in the utterances are tagged as identifiers, entities, or ignored words.

Note that the pattern statement is a template for how the utterance should look in terms of the relative ordering of the identifiers and the entities. As the next step, we consume from the utterances while we simultaneously traverse the pattern statement. If the pattern specification expects an identifier at a particular location, then we skip past the ignored words to locate that identifier in the utterance. We raise an error if we cannot find the identifier.

Similarly, if the pattern statement specifies an entity, we skip past the ignored words and extract the entity from the tokens using the entity-specific parsers. The parsers throw a parse error if they cannot parse the entity correctly. For instance, the entity parser for the criterion in `TrainDecisionTreeClassifier` pattern will reject the utterance “train a decision tree

Algorithm 1: Algorithm for Intent Classification

```

1  input : A user utterance  $ut$  in DataChat CNL, a map  $pm$  that
      maps pattern identifiers to a list of patterns. A single
      pattern identifier can occur in multiple patterns.
2   $tokens \leftarrow \text{Tokenize}(ut)$ 
3   $\text{PatternIdentifierCount} \leftarrow \{\}$ 
4  for token  $t$  in  $tokens$  do
    // only if the token is a pattern identifier
5  if  $t \in pm$  then
6  | if  $|\text{CandidatePatterns}| == 0$  then
7  | | // Take all the patterns for that token
7  | |  $\text{CandidatePatterns} \leftarrow \text{CandidatePatterns} \cup pm[t]$ 
8  | else
8  | | // Only keep the patterns that are common between
8  | | the  $\text{CandidatePatterns}$  and the patterns for the
8  | | token. We hierarchically prune the pattern
8  | | space as we see more and more tokens.
9  | |  $\text{CandidatePatterns} \leftarrow \text{CandidatePatterns} \cap pm[t]$ 
10 if  $|\text{CandidatePatterns}| == 1$  then
10 | // We've found our target pattern
12 | return  $\text{CandidatePatterns}$ 
13 else
13 | // Count the identifier matches for each pattern
14 | for token  $t$  in  $tokens$  do
15 | | for pattern  $pat$  in  $pm[t]$  do
16 | | |  $\text{PatternIdentifierCount}[pat] \leftarrow PC[pat] + 1$ 
17 |  $\text{CandidatePatterns} \leftarrow \text{argmax}_{pat}(pm)$ 
18 | if  $|\text{CandidatePatterns}| == 1$  then
18 | | // We've found the target pattern with the most
18 | | number of identifier that matches the user input.
20 | | return  $\text{CandidatePatterns}$ 
21 | else
23 | | return Error

```

classifier with criterion cross-entropy", as the value "cross-entropy" is not a valid entry accepted by the parser for criterion (should be one of GINI or Entropy).

In both the intent classification and entity extraction phases, we use the lemma form of the word to make the CNL engine robust to morphological variations.

3.3 Auto-suggest

While DataChat's CNL specification reduces ambiguity in interpreting NL, it still leaves the door open for a few other user experience-related challenges:

Utterance Composition: How can the system guide the user so that they can easily compose utterances that adhere to the CNL specification?

Task Capability Discovery: How can the system make its capabilities known to the user so that the user can pick the capabilities they need to solve the problem at hand?

To address these challenges, we couple our pattern specification with an auto-suggest mechanism that guides the user to compose utterances in DataChat's CNL. As the user starts typing into a chat textbox, the auto-suggest mechanism starts to recommend possible completions for the user to choose. These possible completions range from identifiers, keywords in the pattern language to entity values. Figure 3.1 shows the working of auto-suggest in DataChat. When the user begins with an empty input, the auto-suggest mechanism suggests words such as Plot, Record, Remove, Rename, which are words that we use as identifiers in our pattern specifications. Once the user has typed "Plot", auto-suggest deems the suggestions that correspond to the other terms such as "Record", "Remove", "Rename" irrelevant and discards the suggestions. As the user continues to follow the suggestions, the auto-suggest mechanism discards

Plot	a bar chart with x-axis
Record	a boxplot with x-axis
Remove	a bubble chart with x-axis
Rename	a correlation matrix between
Replay	a donut chart using the column
Sample	a geo chart of
Save	a heatmap of the column

Plot	
(a) Initial auto-suggest list	(b) Pruned suggestion list after the user types plot

Figure 3.1: Auto-suggest in DataChat

more irrelevant suggestions. The auto-suggest mechanism successively partitions the space of utterances (tasks) into narrower subspaces until the user arrives at a single valid task specification in DataChat’s CNL.

3.4 Skills as a formalism of task templates

Recall that one of the main characteristics of a DS workflow is that the tasks in the workflow are templated. In Ava, these task templates are represented as code templates. The main restriction of having static code templates is that one code template cannot invoke another code template. This restriction limits the ability to use the task templates to compose more complex workflows. To get around this restriction, we formalize the task templates as *skills* in DataChat. Skills are Python classes consisting of a pattern in DataChat’s CNL that is used to understand corresponding user utterances. The skills also implement logic (in code) that specifies how to perform a task. This logic/code is equivalent to the task-template in

Ava. Skills may also invoke other skills. The execution of a skill must record success or failure status. Skills are logically grouped into skill sets. A few skill sets in DataChat are Data I/O, Data transformation, Natural Language Querying (NLQ), Visualization, Machine learning, Time series analysis, and Graph analysis.

3.5 Modified Client-Server Architecture

Similar to Ava, DataChat also adopts an overall client-server architecture. The initial prototypical version of Ava was a monolith; several components were tied together into a single bundle of software with Jupyter notebook as the backbone of the system. The monolith software design is an anti-pattern in software engineering. Monoliths are hard to maintain, incur high development costs, and are nearly impossible to scale.

The main goal in our system redesign was to break up the monolithic structure of Ava into several independent but synergistic components that interacted with each other to compose DS workflows from NL utterances. To that end, we adopted the microservices architectural pattern in the design of the DataChat platform. Microservice architecture is an architectural design pattern that modularizes an application as a collection of highly maintainable, loosely coupled, and independently deployable services [45]. Figure 3.2 shows the architecture of DataChat. Names with underscores mark the microservices.

3.5.1 The DataChat App Ecosystem

Each user starts their data analysis workflow by launching an application (shortened as "app") in DataChat. Apps are collections of skill sets that help the user in solving a particular data analysis problem. These skill sets can be unique to the app or shared across several apps. For example, an app built for loan analysis can have data loading and data cleaning skill

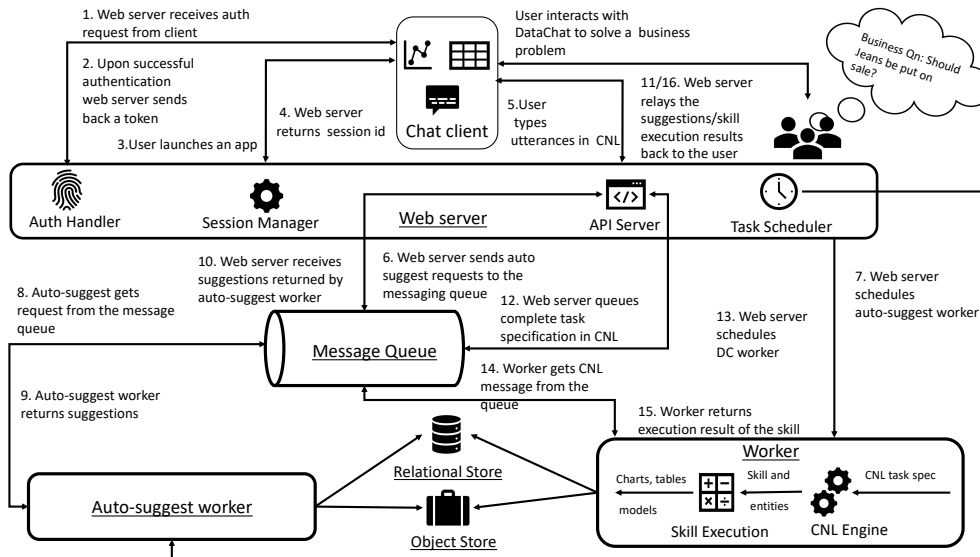


Figure 3.2: The DataChat microservices architecture.

sets, which other applications in DataChat can use.

```

1 "LoansAnalysis": Pattern( 'SHOW the LOANS <
    approval_year> <state>',
2 synonyms={
3 'show ': ['find']
4 },
5 entities={
6 'approval_year': KeywordEntity(
7     Year(), "When should the loans have been
    approved", ["approved"]),
8 'state': KeywordEntity(
9     List(State()), "Which states should the loans
    have been approved in?", ['state'])

```

10

})

Listing 3.4: The LoansAnalysis pattern

The loan analysis application can also have a skill that analyzes loans approved by state (shown in Listing 3.4), which is specific to that app. Apps can target users with different levels of expertise. For instance, an app targeting data scientists may consist of statistical inferencing and ML-related skills that enable them to build models and fine-tune hyperparameters. However, an app targeted for Business Analysis may wrap these specific ML skills into an Auto-ML skill that automates model building and hyperparameter tuning for them. Thus, one can think of DataChat as an ecosystem of data analysis apps (shown in Figure 3.3) akin to the smartphone app eco system, where apps exist for a specific purpose (a weather app only shows weather and a stock app shows stock prices), and the apps leverage the capabilities of the underlying platform.

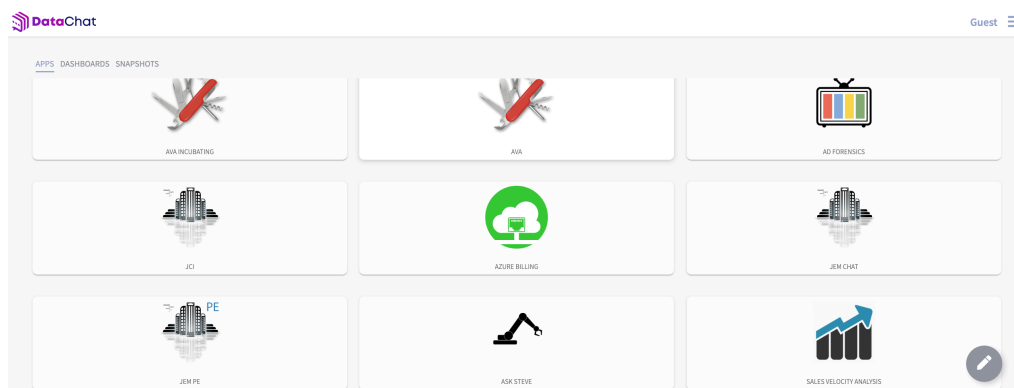


Figure 3.3: The DataChat app ecosystem.

3.5.2 Chat Client

Users launch DataChat apps through a chat client, which is a web application that runs in a browser. When a user launches an app, a session is created for the user. The chat client sends messages typed by the user in a session to the web server through a RESTful API. The client is responsible for rendering charts and tables and exposes functionality to upload data files and download charts as images. The DataChat chat client is implemented using the React JavaScript framework [46]. The module design of the chat client enables it to be easily extended to build tools like dashboards.

Once the user submits an utterance in a session through a web app, the web server receives the utterance and queues the utterance for further processing. The web server also performs other functions like managing user authentication and handling file uploads.

3.5.3 DataChat Worker

Messages sent by the user in a session are queued until a DataChat worker processes the messages. A unit of work in DataChat is an utterance from the user. The worker performs the core functionality of mapping an utterance in DataChat CNL to a task (skill) and executing that task. The worker consists of a core platform that maintains session context, including the conversational history, user credentials, database connections, cone, and other user metadata relevant to the session. The core platform also consists of a CNL engine that can parse utterances in DataChat CNL. DataChat's workers are implemented in the Python programming language to leverage the rich PyData ecosystem of data analytics tools.

Auto-suggest workers Auto-suggest workers aid the user in composing an utterance in DataChat's CNL by suggesting possible completions for a partial utterance. Figure 3.1 shows the working of auto-suggest in

DataChat. When the user starts typing in the chat box, a request is made to the auto-suggest worker to fetch the best possible completion(s) to what the user has typed so far. These completions can either be single word or multi-word phrases. The auto-suggest worker constructs a pattern tree based on the pattern present in each skill of the app. The auto-suggest worker traverses the pattern tree and uses the partial utterance from the user to find the set of next best completions in a greedy manner i.e., if there is only one multi-word completion that matches the users' partial utterance, the auto-suggest engine will fetch the entire phrase instead of just fetching the next word. The auto-suggest worker is aware of the user's session context. The auto-suggest worker can use this context to suggest column name completions or column value completions to the user.

DataChat's workers are stateless. They are not bound to a single user session. Once a worker has executed a skill in a session, the worker persists all session states and objects created through skill execution to external data stores and makes itself available to process the next utterance (which can belong to a different user session).

3.5.4 Data Stores

The DataChat worker persists all session state and objects created through skill execution to external data stores. A skill execution can create various types of objects, and there are three fundamental types: `relations`, `charts`, and `models`. DataChat creates relation objects when users load data from a file, connect to an external database, or run data analysis queries. The results of such operations are stored as tables or views in a database called the **relation store**. Relational objects are Python objects that act as handles to the actual table or view in the relational store. These relation objects are stored in the **object store**. Chart objects are created by executing the visualization skills. The chart objects are stored in the object store. We use Redis [47] as the object store. Invoking ML skills creates

ML models, which are stored on disk. The model object (which is also a Python object) points to the location of the model on the disk.

Objects are stored in the object store as key-value pairs. The object store exposes an API that consists of methods like `get`, `put`, `drop`, and iterators to iterate over its objects. Skills use this API to interact with the object store. The object store is partitioned into namespaces based on the username. Sessions created by the user partition the user namespace into sub-namespaces that are specific to each session. Objects created in a session are stored under the session namespace by default. But skills can also override this by choosing to store objects in the user namespace.

3.5.5 Transactions Over Object Stores

Recall that skills can call other skills, and these nested skill invocations can run arbitrarily deep. Every skill invocation returns a success or failure (uncaught exceptions are skill failures). The object store ensures that any state changes or objects created by failed skill do not linger around after the failed skill exits. The object store achieves this “atomicity” property by committing state changes only on successful exit from a skill and rolling back state changes on failure. State changes made by a skill are written into **tombstones**. These tombstones are not accessible via the main object store API. Thus, if a skill `S2` tries to access an object which is being modified by another skill `S1`, the skill `S2` can only see the old value of the object until skill `S1` returns a success return code. The object store persists the changes made within a tombstone only if the skill execution was successful. Otherwise, changes made within the tombstone are discarded.

3.5.6 Publish-Subscribe Communication Mechanism

The web server and the workers interact through a publish-subscribe mechanism to exchange messages between the user and the worker. For each

session, a pair of messaging queues acts as the communication channel between the web server and the workers. The web server proxies the messages received from a user session (via the chat client) into a messaging queue (incoming). Similarly, the worker sends its responses for an utterance through another messaging queue (outgoing). We use RabbitMQ [48] to manage the creation and deletion of messaging queues in DataChat.

3.5.7 DataChat Scheduler

DataChat maintains a pool of workers to process user utterances and execute skills corresponding to those utterances. Recall that each user interaction with DataChat happens in a session. The web server assigns a unique session id to each user session. Workers in DataChat are not bound to a single user session. Once a worker has finished processing an utterance of one user session, it can process utterances from a completely different user session.

A scheduler that runs in the web server assigns work to the workers. The scheduler maintains a map of worker-to-user session assignments. Once the web server receives a message from the chat client, the server places the message into a *task exchange*. All workers subscribe to the task exchange. The scheduler looks at the pool of workers and assigns an available worker to process the message. If a worker is already processing messages from a session, the scheduler does not assign workers to subsequent messages belonging to the same session. This way, the scheduler guarantees the serial execution of the utterances.

Once a worker receives a message from a user session, it subscribes to the incoming and outgoing message queues for that particular session. The worker starts consuming messages from the incoming session queue one by one until no message remains in the queue. During this action, if the worker wants to send responses to the client, it publishes them to

the outgoing session queue. The worker sends an acknowledgment to the scheduler only after it has processed the message completely. The scheduler can re-assign the message to another worker if it receives no acknowledgment within a given timeframe. Thus, the scheduler ensures that messages are not lost if the worker processing the message crashes.

3.6 Save and Replay Mechanism

Users interact with DataChat in CNL and compose data analysis workflows by invoking skills within a DataChat app. The sequence of utterances specified by the user act as a no-code program documenting the user's thought process and forms a solution template for the business problem that the user is solving. A DataChat user can save and reuse this solution template to solve similar business problems in the future. (They can also share this chat with other users to collaborate.)

DataChat exposes a save mechanism through which the user can save the entire data analysis workflow created within a DataChat app. The DataChat worker logs every utterance made by the user, along with the session state. Persisting a data analysis workflow is as simple as storing the sequence of user utterances. Since failed utterances do not have any persistent effects on the session state, the save mechanism excludes them (safely) from the saved workflow. The workflows saved in this manner can be reused later through a replay mechanism. The replay mechanism sequentially executes the user utterances within the saved workflow. The replay mechanism stops running the saved workflow when it encounters an error while executing an utterance in the workflow. DataChat implements the save and replay mechanisms as skills. We note that that replay mechanism should be atomic, i.e., objects created during a failed replay should not be persistent. By implementing the replay mechanism as a skill, we get atomicity without any additional work due to the properties

of the tombstone mechanism.

3.7 DataChat as a Platform

As we described in Section 3.5.3, DataChat exposes a core platform API that developers can use to create new skills and apps. This core API includes functions that allow for one skill to invoke another skill in DataChat's CNL. This mechanism allows for more complex skills to be composed from simple skills.

In addition to the core platform API, users can access the capabilities of DataChat via a REST API. Through this API, users can issue utterances and retrieve objects produced in DataChat. The REST API makes DataChat accessible to interfaces outside the DataChat software. Third-party developers can develop custom user interfaces or integrate DataChat functionality into their products by leveraging the DataChat API.

3.8 Scalability and Deployment

All the microservices in DataChat are containerized as Docker containers [37]. Containerizing DataChat's microservices gives DataChat the ability to be seamlessly deployed on any hardware platform or operating system without having to worry about the underlying platform's capabilities. DataChat's containers are orchestrated using the Docker Compose framework [49]. Through Docker Compose, DataChat can seamlessly scale up and scale down its microservices within a single machine and across multiple devices. The design of DataChat also makes it possible to scale microservices independently based on the workload. For example, one can selectively choose to scale-up the worker and auto-suggest worker to handle more concurrent users.

3.9 Conclusion

In this chapter, we present the system design and architecture of DataChat, an improved version of Ava. We take a microservices architecture approach to modularize DataChat into several microservices and seamlessly scale the microservices by containerizing them using Docker and docker-compose.

We also take a stateless/serverless approach in designing our microservices. In a conversation-based platform like DataChat, the system is likely to spend most of its time waiting for the user to respond. The user can execute an utterance and then spend a few minutes examining the resulting chart or table. If workers/resources are bound to a user (i.e., the platform holds the state of the user in memory), the worker will stay idle until the user issues the next utterance, and thus compute and storage resources will be utilized in a suboptimal way. A stateless approach frees up the worker from waiting for the user thus making this approach more resource-efficient.

We propose a more flexible CNL, implemented as a pattern language, that allows the user flexibility in composing CNL utterances. Our current CNL framework provides a first step towards making DataChat accessible to a broader audience through the use of uncontrolled NL.

4

DataChat: Ava Revisited

In the previous chapter, we proposed modifications to Ava’s CNL to cater to a broader set of analytics tasks and formalized Ava’s CNL via a pattern language. We also presented improvements to Ava’s system architecture, implemented as a new system called DataChat. This chapter focuses on DataChat’s enhancements for model selection, model training, Automated Machine Learning (AutoML), and model inferencing. We introduce the notion of an ML pipeline that serves as the abstract representation of the ML task specified using Ava’s modified CNL specification. Finally, using the new DataChat platform, we evaluate the efficacy of DataChat’s CNL and the ML Pipeline framework for training models via a user study, and we present our results.

4.1 ML Pipeline

In a typical ML workflow, data scientists start with a set of data points (the training dataset) to identify and train a good predictive model for their task. Often, this raw data cannot be fed to the ML algorithms directly. A data scientist may need to pre-process the data, fill missing values, encode categorical columns, engineer and select features to make the data suitable for ML. After these steps are carried out, the data scientist must

then perform algorithm selection, cross validation, and hyperparameter optimization to obtain a good quality predictive model.

It is common for data scientists to develop models in a notebook-like environment such as RStudio or Jupyter notebook. In such cases, the code for various stages of the model training workflow is likely written in a single notebook. These models are trained and productionized by bundling the code in the notebook into a single code package. The end product, in this case, is an ML model suitable for the task. This process, however, is ad-hoc, challenging to maintain and scale because the workflow developed in the notebook is difficult to repeat and document. A code-monolith (notebook-based) approach to training ML models also makes retraining, measurement, and testing of models cumbersome. Additionally, the code-monolith approach makes it hard to detect and isolate errors during the model building process.

In contrast to the code-monolith way of training and deploying ML models, an *ML pipeline* splits the ML workflow into independent, reusable modular components. These independent components can be assembled to train and deploy a model. An ML pipeline is a means to codify and automate the workflow that is required to produce a machine learning model. An ML pipeline consists of the various components of the ML workflow and the map of how these components are combined to create the model. The end product here is the end-to-end model training pipeline, which can be reused and maintained.

4.1.1 ML Pipeline Advantages

Reusable sub-components In the ML pipeline, each stage of the model training workflow is abstracted into an independent component. Thus, when a new model needs to be trained, one has the flexibility to selectively pick the pipeline components necessary for building that particular model.

Subcomponents of the ML pipeline can be shared across users in the organization.

Easy to scale The use of ML pipeline to train models within an organization provides a global view of the ML model training workflows within the organization. This global view provides opportunities for cross-ML workflow optimizations. For example, one can identify specific parts of the pipeline that are heavily used throughout the organization. These commonly used pipeline components can be cached and reused to improve system efficiency and human productivity.

Easy to version and update With a single implementation of the ML pipeline components being reused across various ML model workflows, it becomes easier to version and update the pipeline components. Updating the implementation of a single pipeline component will update all the workflows that reuse that pipeline component.

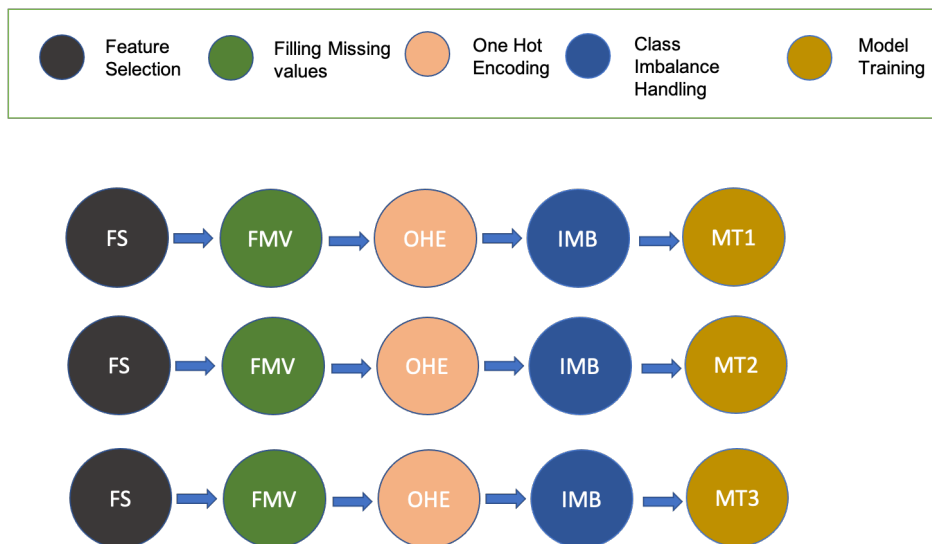


Figure 4.1: Multiple analysis without ML pipelines

The case for ML pipelines Consider the churn prediction task from

Chapter 2. An ML workflow for training churn prediction models could consist of steps such as feature selection, filling missing values, one-hot encoding, class imbalance handling, and model training.

Figure 4.1 shows how training of multiple models would look like with monolithic structures. The entire end-to-end workflow for training the ML model is repeated for every model training iteration.

Contrast this with the ML pipeline approach shown in Figure 4.6. It's easy to swap out the ML model (algorithm) without breaking the whole workflow while reusing the results of the common steps of the workflow across every model training iteration.

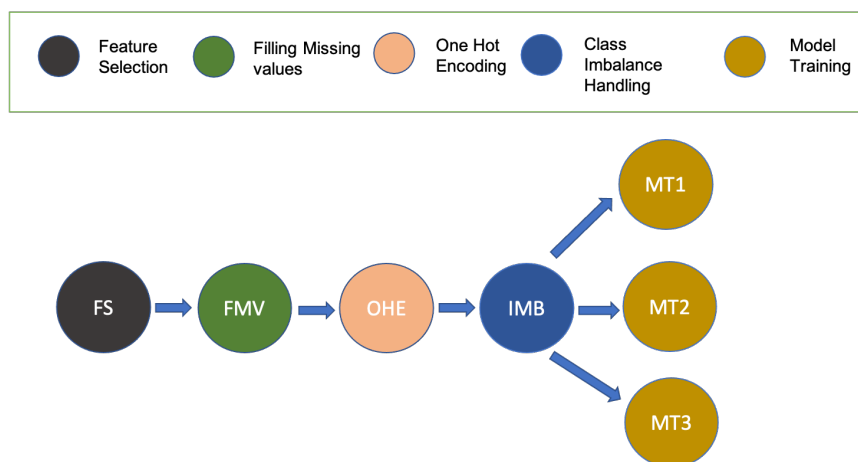


Figure 4.2: Multiple analysis with ML pipelines.

4.2 Design Principles

We designed the DataChat's ML pipeline framework to be more flexible than Ava's ML infrastructure while retaining its ability to be composable using NL utterances. The ML pipeline serves as an abstract representation of the ML task specification, which maps the ML task specification in

DataChat's CNL to the various stages involved in ML model training workflow. In this section, we describe the design principles behind DataChat's ML pipeline infrastructure.

Dataflow graph of logical DS workflow stages The ML pipeline graph is a Directed Acyclic Graph (DAG) representing the flow of data between the different logical stages of the DS workflow. The advantage of using a DAG is that every node executes only once per execution instance of the pipeline. This makes the pipeline simpler to define, understand, and debug.

Framework agnostic The ML pipeline graph is independent of the underlying DS framework/package (e.g., scikit-learn), which often provides the concrete implementation of that component in the ML pipeline. In the monolithic way of training ML models, a single programming language and ML framework is used for training the ML model. But in the ML pipeline framework, different parts can be written in different languages, each using its own framework. This is a key strength when scaling model training and deployment since it allows components of the pipeline to be reused, regardless of language or framework .

Lazy execution Execution of the ML pipeline in DataChat is done in two phases: The first phase defines the ML Pipeline graph with the different ML model training workflow components and the interconnection between them. The second phase involves the execution of the pipeline components represented by the pipeline nodes in the ML graph. Deferring execution of the ML pipeline graph offers the flexibility to incorporate framework choices at run time instead of choosing the framework upfront. For example, one could start the ML model building process with an initial dataset of a few hundreds of MBs that can be comfortably loaded using pandas. However, the dataset may have many categorical columns, which will lead to an explosion in the size of the dataset during the one-hot encoding phase. It may be inefficient at this point to use scikit-learn as

the framework of choice for model training. Due to the lazy evaluation nature, the ML pipeline framework can detect the explosion in the size of the dataset and switch to a framework like Dask [50] as the downstream framework for model training.

Reusability of intermediate results The execution results of the pipeline components can be cached and reused to warm start other model training pipelines/workflows. For instance, in an ML pipeline that uses one-hot encoding to train a decision tree model, the one-hot encoded dataset can be cached. A new ML model that is trained on the same dataset can reuse the cached result.

Parallelization Independent components of the ML pipeline graph can be executed in parallel, resulting in faster model training times.

4.2.1 Execution Model

DataChat uses a single DAG to represent the different components of the ML workflow and the data dependencies between the workflow components. The nodes in the ML pipeline graph are stateless. This approach makes it easy to execute independent components in parallel. The execution of these separate components can be scaled out to other devices if necessary.

4.2.2 Constituents of the ML Pipeline Graph

Pipeline node We refer to the vertices in the ML pipeline graph as pipeline nodes. An ML pipeline node is an abstract representation of the logical states (e.g., feature selection, cross-validation) of an ML model training workflow.

Pipeline edges represent the flow of information from one pipeline node to another. Information that flows across the edges can be datasets

created as a result of the execution of the pipeline node, or other metadata that is produced as the result of the execution of upstream pipeline nodes.

ML knowledge base This is a Data Science (DS) knowledge base that encodes heuristics and common rules of thumb used by data scientists for training models.

Node executor Each pipeline node has a node executor associated with it. The main purpose of the node executor is to instantiate a concrete implementation of the pipeline node and execute that implementation. To do so, the node executor can use the metadata from the execution of its parent nodes. Additionally, the node executor interfaces with the ML knowledge base. The node executor can leverage the knowledge base to make decisions regarding the choice of framework to execute the current pipeline node. The node executor is responsible for caching the results of the current pipeline node's execution. The node executor stores the node's output in a key-values store, and passes the keys to the downstream nodes, which are used by the downstream nodes to access the relevant information from the parent nodes.

Pipeline executor The pipeline executor is the central coordinator that is responsible for initializing the ML pipeline graph and invoking the node executors. The pipeline executor maintains a global view of the execution states of the various nodes in the pipeline graph. The pipeline executor can also make decisions regarding execution of downstream pipeline node based on the current execution state. For example, if the executor observes that no missing values were detected in the data, it can skip the execution of the fill missing values pipeline node. Note that the fill missing values pipeline node is not removed from the pipeline graph. Retaining the missing value node in the pipeline makes the pipeline robust to missing values that could occur during future training iterations or predictions.

Pipeline graph execution In DataChat, the "train model" CNL is used to instantiate the pipeline executor. The pipeline executor creates pipeline

nodes and initializes the pipeline nodes with information extracted from the “train model” CNL. At the start of the execution, the ML pipeline executor topologically sorts the ML pipeline graph and begins invoking the node executors.

4.3 AutoML in DataChat

A typical ML workflow involves various steps such as filling missing values, encoding categorical columns, feature engineering, feature selection, model selection, hyperparameter tuning, cross-validation, and model training. The ML workflow is iterative, and it involves human experts trying out various tools, techniques, and algorithms in the different stages of the ML model building workflow to develop a good quality ML model. Each of these steps may be challenging, resulting in significant hurdles to using machine learning. Automated Machine Learning (AutoML) is the process of automating the iterative tasks involved in machine learning model development. AutoML dramatically simplifies the steps involved in training an ML model for non-experts.

DataChat’s AutoML functionality allows users to train a variety of Gradient Boosting (GB) models ([51–53]) for classification and regression tasks. Gradient boosting methods are a class of algorithms that have recently grown in popularity [54]. Since their advent, these algorithms have dominated Kaggle competitions for classification and regression tasks on tabular data and have become the most popular algorithms on competition sites like Kaggle [51]. We’d like to note that restricting the class of models explored by the AutoML framework is a design choice rather than a limitation of our framework. By limiting the space of models to the high-performing class of GB models, we reduce the effort (and time) spent by the framework in finding a high quality model while retaining the ability of the framework to generate high-quality models. The AutoML

framework's hyperparameter tuning interface can also be used to train and tune specific models (e.g., Decision Trees, Logistic Regression), which are not included by default in the framework's model search space.

DataChat's AutoML framework is built on the ML pipeline infrastructure, making it easy to extend the framework with new models. The "Analyze" utterance in DataChat's CNL invokes DataChat's AutoML functionality. Users can "Analyze" a dependent variable using an optional list of independent variables. Under the hood, Ava's AutoML framework creates and executes ML pipelines for the space of models that it explores. Common intermediate results from the execution of pipeline nodes are reused across model search pipelines. The AutoML framework also taps into the knowledge base to efficiently search through the model space. For example, suppose the AutoML framework detects many categorical features in a dataset that could cause an explosion in dataset size due to one-hot encoding. The framework could decide to explore only Catboost models as these models as that algorithm does not require categorical columns to be one-hot encoded. At the end of the model exploration process, the AutoML framework evaluates the models using cross-validation and returns the best scoring model. The scoring metrics are automatically chosen based on the type and distribution of the dependent variables. Users can also override the AutoML framework's choice of metrics.

Listing 4.1 shows the pattern for DataChat's AutoML CNL.

```

1 "AnalyzeDataset" : Pattern('ANALYZE <
    dependent_variable> <independent_variables> <
    exclude_features> <enabled_optimizations> <
    disabled_optimizations> <label_weight_ds>
    setting the value of <timeout>'),
2 synonyms={
3 'train': ['build'],
4 },

```

```
5 addnl_ignore=['model', 'and'],
6 entities={
7     'dependent_variable': PositionalEntity(
8         NonRepeatingColumn(), "Which column
9         would you like to analyze?"),
10    'independent_variables': KeywordEntity(
11        List(NonRepeatingColumn()),
12        "Which columns would you like to
13        use as the independent variables?", ['using']),
14    'exclude_features': KeywordEntity(
15        List(NonRepeatingColumn(
16            autocomplete_placeholder='columns to exclude')),
17        "Please provide a comma separated
18        list of column names you would like to exclude "
19        "from Analyze.", ['excluding']),
20    'enabled_optimizations': KeywordEntity(
21        ManyOf(candidates=
22            get_analyze_optimizer_enablers(), separators=',,
23            ),
24        "Which optimizations do you want to
25        enable for this Analysis?",
26        ['with']),
27    'disabled_optimizations': KeywordEntity(
28        ManyOf(candidates=
29            get_analyze_optimizer_disablers(), separators=',,
30            ),
31        "Which optimizations do you want to
32        disable for this Analysis?",
33        ['without']),
```

```

24         'label_weight_ds': KeywordEntity(
25             Dataset(autocomplete_placeholder='
dataset mapping used to weigh imbalanced data
points'),
26             "Please provide the name of the
dataset with which to differentially weight data
points based on target class.",
27             ['weighting with']),
28         'timeout': KeywordEntity(
29             Number(),
30             "What is the maximum time in
seconds you would like to wait "
31             "for Analyze to finish?",
32             ['timeout']
33         ),
34     })

```

Listing 4.1: Analyze's CNL pattern specification

4.3.1 Feature Transformations

DataChat's AutoML framework automatically infers the types of features in the data. The framework uses these inferred types to explore various feature transformation operations.

Automatic feature discretization Feature discretization or feature binning is a commonly used feature engineering technique to smoothen out the noise in continuous columns. Discretizing continuous columns has the added advantage of interpretability. For example, a continuous variable such as Age can be understood better when discretized into buckets such as infants, youth, and adults. The AutoML framework discretizes con-

tinuous variables in relation to the target variable using a mixed-integer program formulation for optimizing the bins [55].

Automatic extraction of temporal information Date-time fields are a common occurrence in real-world datasets. Extracting temporal information from the date-time fields as features helps the model capture seasonal trends and patterns in the data. DataChat’s AutoML framework automatically detects periodicity in the data. The framework then extracts temporal information from the date field.

Feature pruning DataChat’s AutoML framework automatically identifies and eliminates features that are low signal by applying certain heuristics. This includes removing fields like id columns, where most of the values are unique. Additionally, the AutoML framework detects and eliminates variables that are not highly correlated with the label. Low-correlation feature elimination is only applied to linear models such as logistic regression. The AutoML framework also detects feature pairs that are highly correlated and eliminates one feature from the pair.

4.3.2 Class Imbalance Handling

An imbalanced classification problem is where there is a skew in the distribution of the known classes in the dataset. Class imbalances pose a significant challenge for modeling as most ML algorithms assume an equal representation of the predictor variable in the dataset. Class skew results in models that have poor predictive performance, specifically for the minority class. Often, the minority class is more important (e.g., fraud detection), and therefore the problem is more sensitive to classification errors for the minority class than the majority class. DataChat’s AutoML framework supports the following strategies for handling imbalanced classes:

Oversampling One approach to address skew in class labels is to augment the minority class with additional samples. The naive approach is

to duplicate the minority samples, but this approach does not provide any new information to the model. Instead, the minority class can be augmented by synthesizing artificial examples from the existing examples – a technique known as SMOTE [56].

Class weighting is an alternate approach to handling imbalanced classes is to use class weighting in which the minority classes are assigned a weight significantly higher than the majority class. This informs the model that the cost of misclassification of the minority class is higher than the cost of misclassification for the majority class and forces the model to pick up signals from the minority class.

Both class weighting and oversampling are implemented as ML pipeline nodes. The AutoML framework uses oversampling as the default class imbalance handling mechanism.

4.3.3 Explanations

The state-of-the-art models explored by the AutoML framework are highly accurate, robust to noise in data, and work well on data from a wide variety of domains. On the flip side, these models are complex and do not offer insight into their inner working mechanisms. The inherent complexity of the models coupled with their opaqueness impairs the ability of data scientists to make decisions based on the results of the model. These issues underscore the necessity of keeping humans in the loop via explanations that can equip them to better understand the models and to make better decisions [57]. The AutoML framework generates the following explanations:

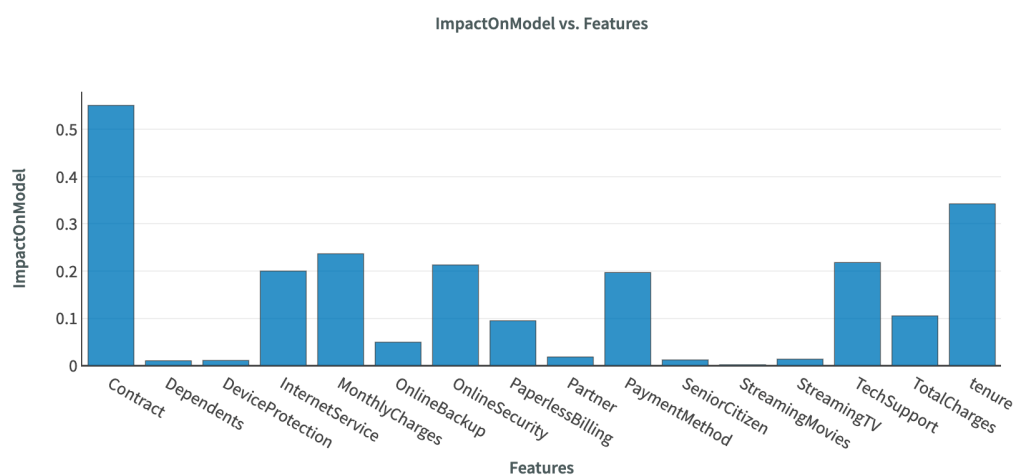


Figure 4.3: Feature importance metrics generated by DataChat’s AutoML for the churn prediction task

Feature importance values indicate how much each feature in the model contributed to a certain prediction made by the model per instance. SHAP [58] is a method that uses a game-theoretic approach to generate feature importances for each instance in the training dataset. The AutoML framework generates model-level feature importances by taking the mean absolute value of the SHAP values per feature, per instance of the training data.

Figure 4.3 shows the feature importance metrics generated by DataChat’s AutoML framework using the SHAP framework. These feature importance scores provide insight into the most important independent

variables that influence the dependent variable. Analysts can use the important features to train other models or to inform downstream analytics tasks.

Data Explanations complement model explanations in that they seek to generate interesting visual explanations (charts) of the dependent variable, solely using the data. These visual explanations are generated by matching the independent variables (specified during model training) with the chart specifications of different chart types from a chart library. These visual explanations are generated by matching the independent variables specified during model training and the specifications (visual encodings and supported data types) of various chart types from a chart library and by picking the best chart that fits the data. The following steps outline the process involved in generating data explanations:

- The type of dependent variable (discrete or continuous) is first used to prune the space of possible charts in the chart library.
- Each chart in the chart library is associated with a specification that defines the mandatory and optional visual encodings channels, e.g., x-position, y-position, size, color, and the data type of the fields suitable for that visual encoding channel.
- Then each chart in the pruned space of charts is checked for plausibility. A chart is plausible only if its specification can be satisfied by a subset of the independent variables specified during model training.
- The plausible charts are ranked based on their information content (the more important features used in the chart, the greater is its score), and the top-k charts are displayed to the user.

Accompanying each visual data explanation, is a description of the steps taken by the AutoML framework, to generate that data explanation (Fig-

ure 4.5). Thus, the AutoML framework makes the insight generation process transparent by providing meta-explanations.



Figure 4.4: Data explanations generated by DataChat’s AutoML module for the churn prediction task

Figure 4.4 shows the auto-generated insights comparing the contract and tenure variables’ influence on the dependent variable Churn. From the chart, one can infer that the churn rate is the highest in the month-to-month bucket and lowest in the two-year bucket and that the longer the customer stays irrespective of contract type, the less likely they are to churn.

This chart was created using the following steps:

- First, I distributed tenure into several buckets, each with size 20, and named the new column tenureInt20
- Then, I computed the count of records for each Churn, Contract and tenureInt20 calling the output column CountOfRecords
- Finally, I plotted a bubble chart with Contract as the x-axis, tenureInt20 as the y-axis, CountOfRecords was used to set the size of the bubble, the bubble color was set using Churn

Figure 4.5: Steps for generating the data explanation in Figure 4.4

Pipeline Reports The AutoML skill generates a detailed report listing all the stages of the ML pipeline, that were executed to train the model. This report includes details on the type of data imputation carried out on the features, the encoding performed on the target variable, the feature transformation steps, the space of hyperparameters searched and the parameters of the final model.

pipeLineStage	message
Impute Features	Features list : Contract, Dependents, DeviceProtection, InternetService, MonthlyCharges, OnlineBackup, OnlineSecurity, PaperlessBilling, Partner, PaymentMethod, SeniorCitizen, StreamingMovies, StreamingTV, TechSupport, TotalCharges, tenure. No missing values found in features.
Impute Label	Selected label : Churn. No missing values found in the label.
Autobinning	Autobinned columns: . Bin Edges: {}
Encode Features	Detected categorical_features are Contract, Dependents, DeviceProtection, InternetService, OnlineBackup, OnlineSecurity, PaperlessBilling, Partner, PaymentMethod, SeniorCitizen, StreamingMovies, StreamingTV, TechSupport. No encoding carried out for categorical features.
Encode Label	The label was LabelEncoded for the detected classes : No, Yes.
Train Model	model_type : catboost_classifier. Hyper-parameter search space : {'max_depth': 'range(3, 16)', 'n_estimators': 'range(10, 100)', 'eta': [0.01, 0.05, 0.5]}. Optimal Model parameters : {'max_depth': 3, 'n_estimators': 96, 'eta': 0.05, 'silent': False, 'od_type': 'Iter', 'od_wait': 19, 'boosting_type': 'Plain', 'bootstrap_type': 'Bernoulli', 'subsample': 0.66, 'max_ctr_complexity': 2, 'random_state': 42, 'objective': 'Logloss', 'loss_function': 'Logloss'}. Model scoring metrics :['accuracy', 'auc']. Model_scores : AUC : 0.76, Accuracy : 77%.

Figure 4.6: Pipeline report generated by DataChat’s AutoML framework provides transparency into the model training process

The pipeline report is an explanation of DataChat’s AutoML model generation process. It provides insights into the various steps taken by the AutoML framework to arrive at the end model. Expert users can peruse the pipeline reports and provide feedback to the AutoML framework by adjusting the model search space via the parameters in DataChat’s AutoML CNL. Thus, pipeline report explanations and DataChat’s AutoML CNL specification provide a mechanism to incorporate human expertise and feedback back to the ML process.

4.4 Experiments

We conducted a user study to test our hypothesis that DataChat’s CNL coupled with DataChat’s ML infrastructure significantly improves the productivity of its non-expert users, with only a small learning curve. In this study, we asked users to complete a Kaggle task using both DataChat and Python packages and measured various metrics of productivity.

Participants We recruited graduate students from the Computer Sciences Department at the University of Wisconsin-Madison and data analysts from the industry for this experiment. All participants in this group had exposure to machine learning and data science through their coursework or their employment.

Task and data description The task that we chose for this study was one of predicting customer churn for a telecommunications firm. The objective of the task was to develop a model to classify which customers are most likely to refrain from renewing their contract when it ends. Participants had to attempt six Exploratory Data Analysis (EDA) questions before they trained models to predict customer churn. We laid out the EDA questions in increasing order of difficulty. We obtained a dataset consisting of historical details and account usage information for customers of the telecommunications firm, along with labels that indicate whether or not they renewed their contract. These labels, however, were skewed in that only 1869 of the 7043 customers had terminated their contract. We sampled a subset consisting of 4921 customer records and provided the sample as training data to the participants. The remaining 2122 customer records were used as a test set to evaluate the models developed by the participants.

All participants used a Linux machine with 4 CPUs and 16GB of memory. Each participant performed this task once using Ava and once using the Python packages (pandas and scikit-learn). We limited the duration

of the task to 60 minutes for the DataChat task. For the Python task, participants were allowed to take as much time as needed to complete the task. Participants were instructed to explore as many models (combination of learning algorithm, features, and hyper-parameter choices) as they could within this duration of the study. They were free to use any documentation, tutorials, or code samples from the Internet to help them complete the task. Participants were also provided with a DataChat reference manual. More details about the dataset and the exact task description are available at [59].

4.4.1 Results

In this section, we discuss the results of our user study.

4.4.2 Productivity Improvement

We used the Task Success metric from Google’s HEART framework [60] to measure productivity improvement. Specifically, we measure:

- the time taken to train the first model,
- the number of models trained during the study,
- the time taken to answer the EDA questions, and
- the number of EDA questions attempted.

Note that all time measurements include the actual processing time for the analytics steps (within the underlying ML and the visualization libraries), in addition to the time spent by participants in manually proceeding through the steps in the analysis task. First, we consider the time taken by the participants to train their first working ML model. The distribution of the results using this metric is shown as violin plots in the Figure 4.7.

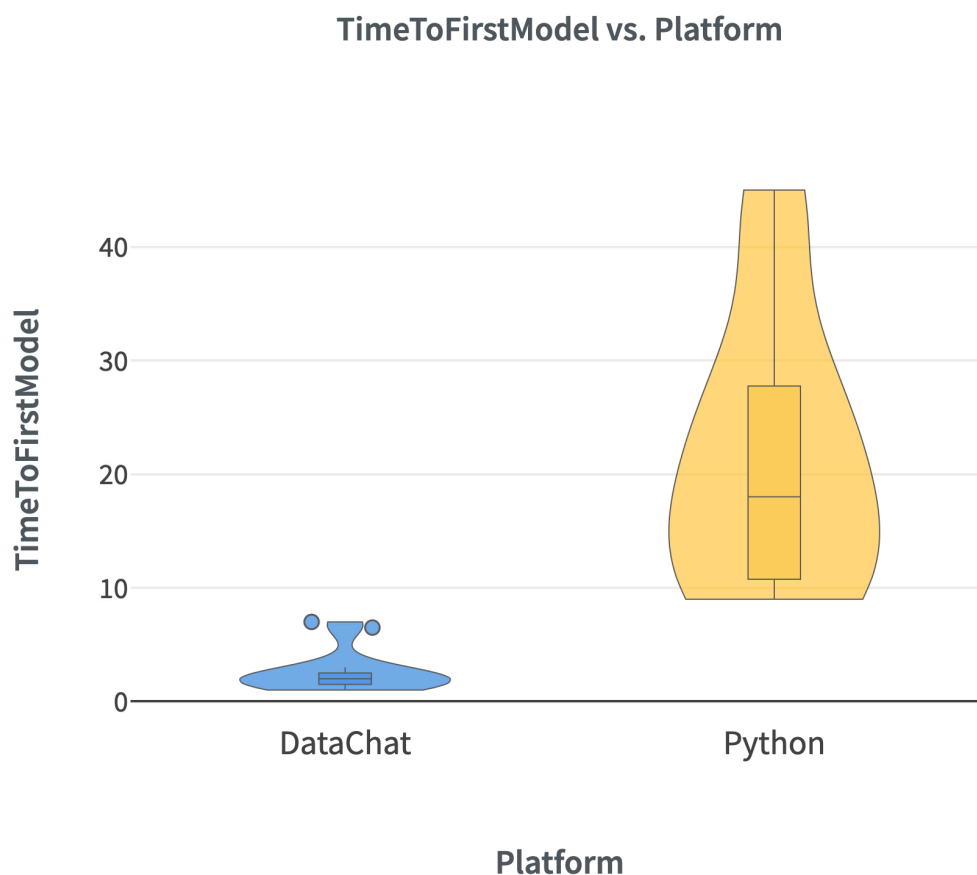


Figure 4.7: Distribution of time taken by participants to complete the first ML model using DataChat and Python.

Twelve out of the fifteen participants were able to complete their first model using Python. The average time taken to train the first model was 21 minutes. On the other hand, all 15 participants completed their first model using DataChat within 2 minutes. Thus, we observed an improvement of more than 10x on this productivity metric.

Further, the average time taken by the users to train the first model (on a smaller dataset) using the Ava prototype was 7 minutes. Thus, there is a

3x improvement in the time taken to train the first model in DataChat over Ava. We attribute this gain in time largely to the conciseness of DataChat's CNL and the auto-suggest functionality that makes it easier for the user to compose DataChat's CNL, which reduces the overall time spent by the user in DataChat's conversational loop (Figure 2.3)

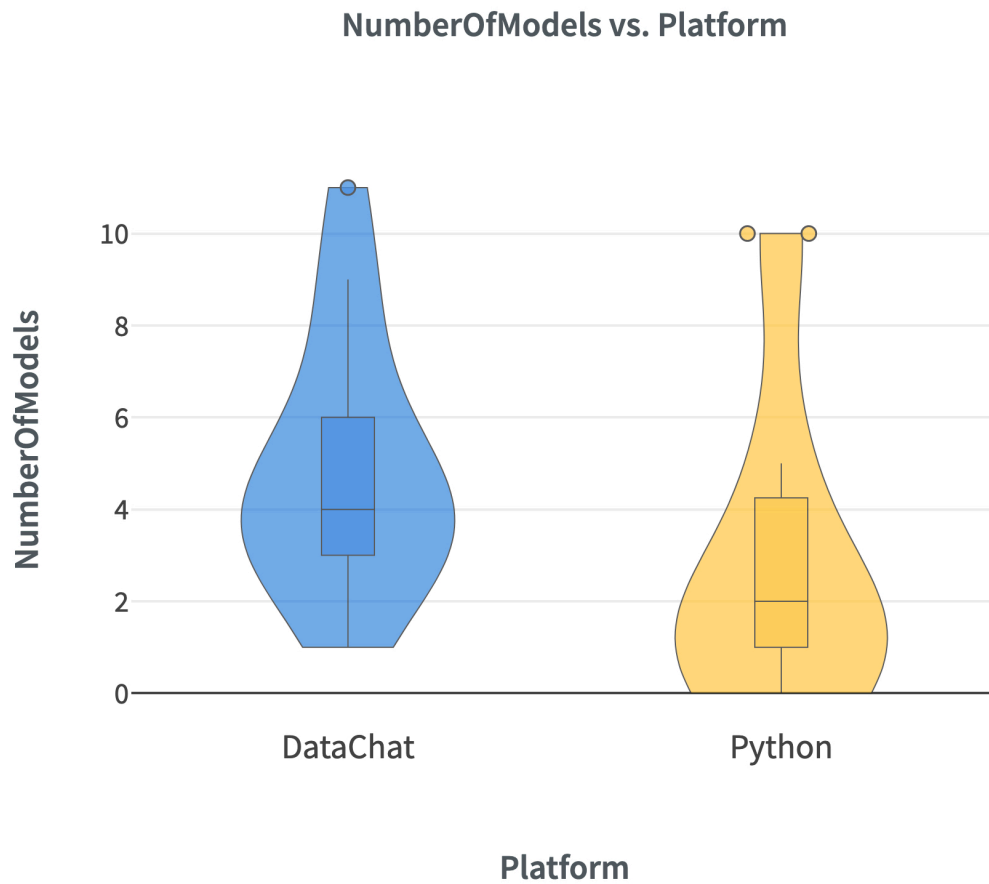


Figure 4.8: Distribution of the number of models trained by participants using DataChat and Python.

We also consider the number of models that the participants could train during the study for each approach (Figure 4.8). Only eight participants

were able to train more than one model using Python packages. On the other hand, every participant was able to explore at least three models using DataChat, and on average, each participant explored six models. Thus, DataChat showed a significant improvement on the number of models trained metric.

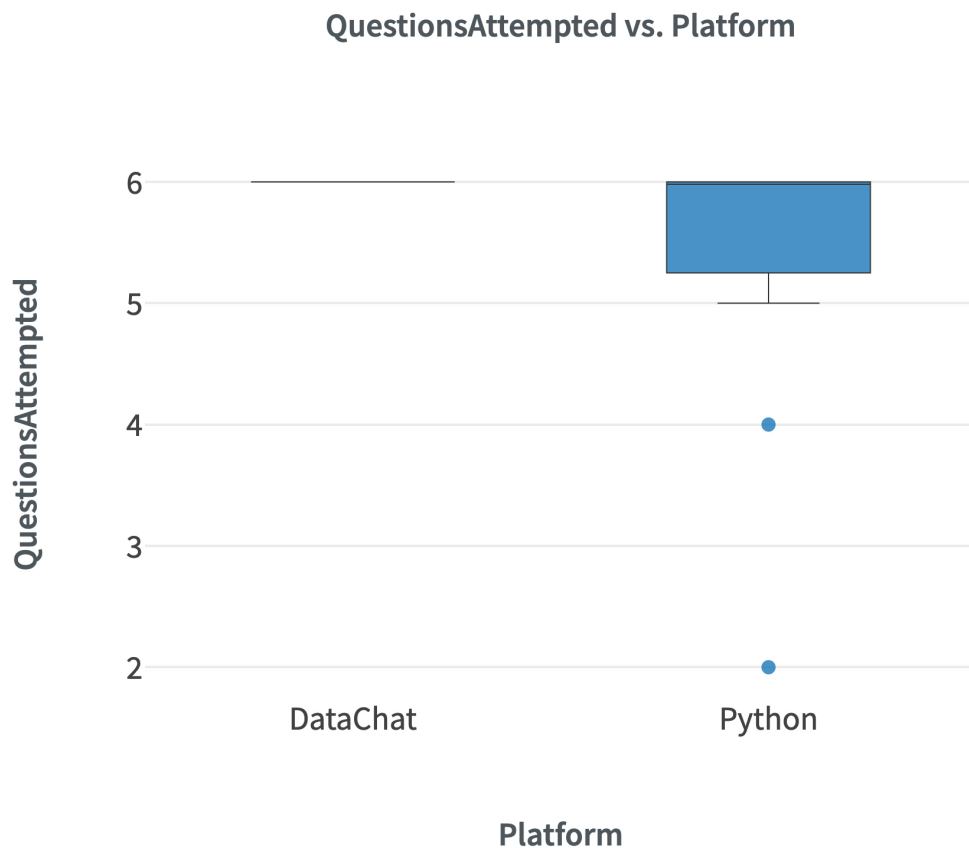


Figure 4.9: Distribution of the number EDA questions attempted using DataChat and Python.

Exploratory Data Analysis For this analysis, we consider that a user successfully attempted a question only if they either had typed an utter-

ance in DataChat or had written Python code to solve the question. All participants were able to attempt all six EDA questions using DataChat, whereas only 11 users were able to attempt all six exploratory questions using Python (Figure 4.9).

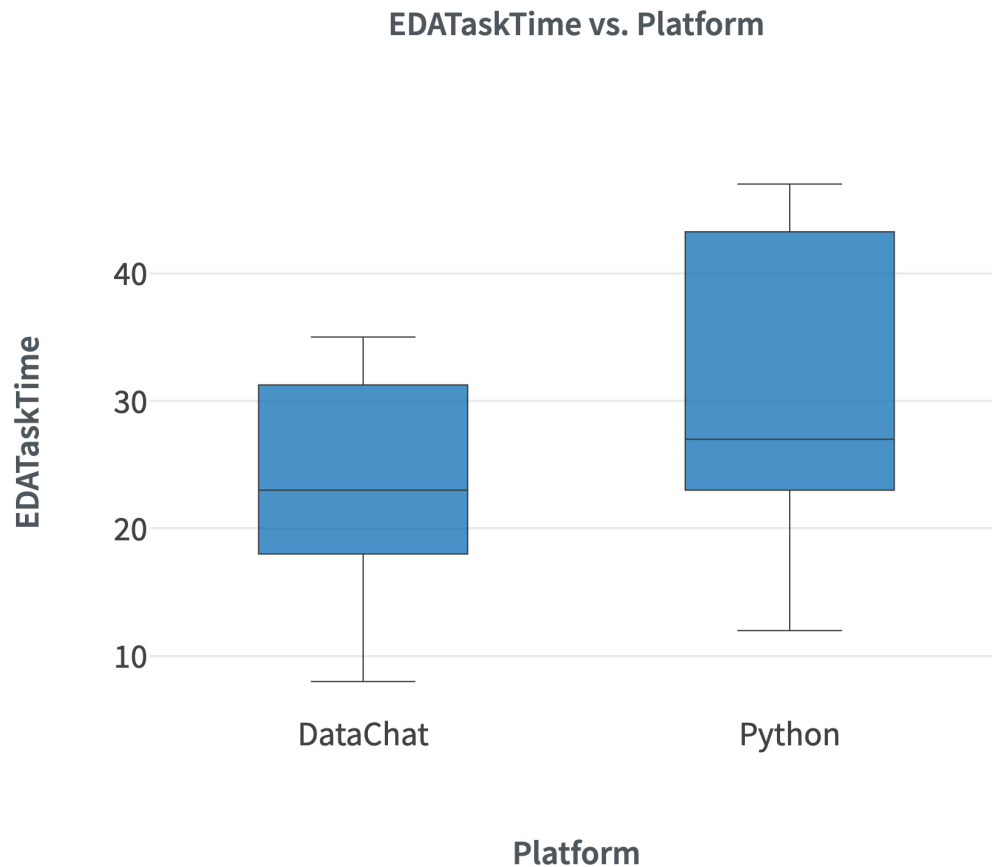


Figure 4.10: Distribution of the time taken to answer EDA questions using DataChat and Python.

Figure 4.10 compares the distribution of the time taken by the participants to complete the EDA task using DataChat and Python. Participants averaged 30 minutes to complete the EDA questions using Python. On

the other hand, all 15 participants were able to attempt all six questions in DataChat. They took an average time of 23 minutes to complete the EDA task.

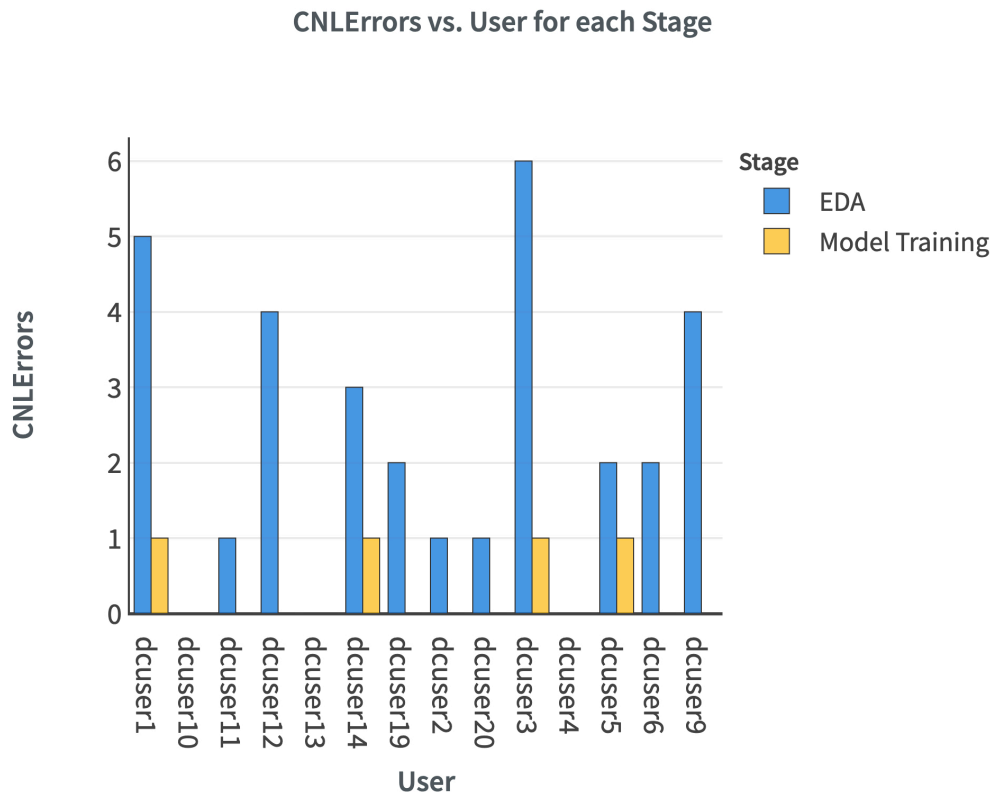


Figure 4.11: Comparison of CNL errors made by users during the initial and later stages of the user study. Users first solved the EDA task followed by the modeling task.

During the EDA section of the study, we observed that users often referenced DataChat’s documentation and the built-in manual to identify the correct CNL utterance for the EDA tasks. We also observed that users typed in utterances that DataChat did not understand as they were not familiar with the auto-suggest mechanism. We believe that these two

factors contributed to participants taking additional time to complete the EDA task.

As the study progressed, participants made fewer errors with DataChat's CNL as they became comfortable with the auto-suggest mechanism. This reduction in errors as users progressed through the study indicates that users will become more productive with DataChat as they acclimatize themselves to the platform.

Training Participants of the study were given a choice of viewing a 15 minute training video or participating in a 10 minute training exercise to familiarize themselves with DataChat. The training focussed on the usage of the auto-suggest mechanism to compose utterances in DataChat's CNL and the UI menus which would be helpful to the participants for the study.

Learning Curve Of the 15 participants in the study, eight were randomly chosen to perform the task using Python packages before doing so with DataChat (we denote this group using the label Group-PD). The remaining seven were asked to perform the task using DataChat before doing so with Python (denoted as Group-DP). Comparing the productivity differences between these groups using the two approaches allows us to reason about the learning curve.

Figure 4.12 shows the distribution of the time taken to train the first model across different task groups. When performing the task using Python packages, the eight participants in Group-PD (who used Python before DataChat) took, on average, 20 minutes to train their first model. On the other hand the 7 participants in Group-DP (who used DataChat before Python) took 3 minutes to train their first model. Participants in Group-PD showed a negligibly small 1 minute improvement (2 minutes) to train the first model using DataChat. Further, we saw only a negligibly small difference in the average number of models trained by the participants using DataChat in the two groups.

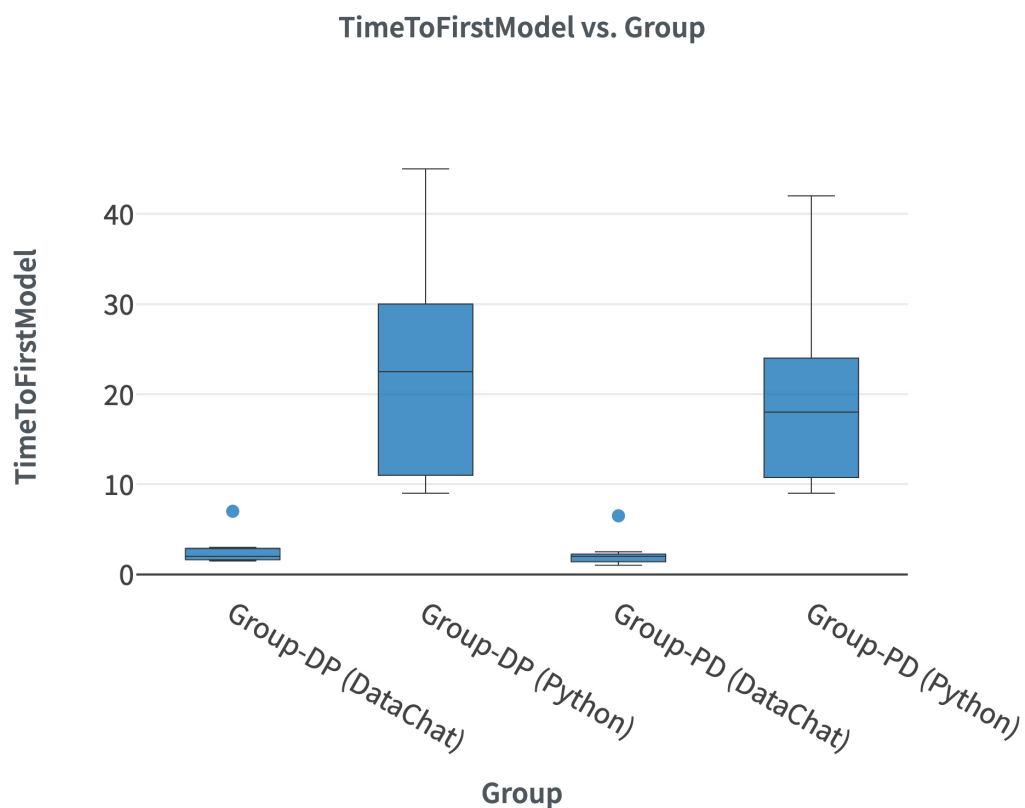


Figure 4.12: Distribution of the time taken by the task groups to train the first model.

We consider these observations as indicating that users, with minimal training can be nearly equally productive with DataChat regardless of their prior experience in or familiarity with the data science process.

We observed that the average number of EDA questions correctly solved by the participants using DataChat (5) was greater than the average number of correct solutions using Python (4). Upon examining DataChat's logs, we found that 59% of the errors (Figure 4.13) were because the users submitted an incorrect CNL utterance to DataChat leading to an incorrect result. We believe that such errors can be avoided by providing more

training to the users.

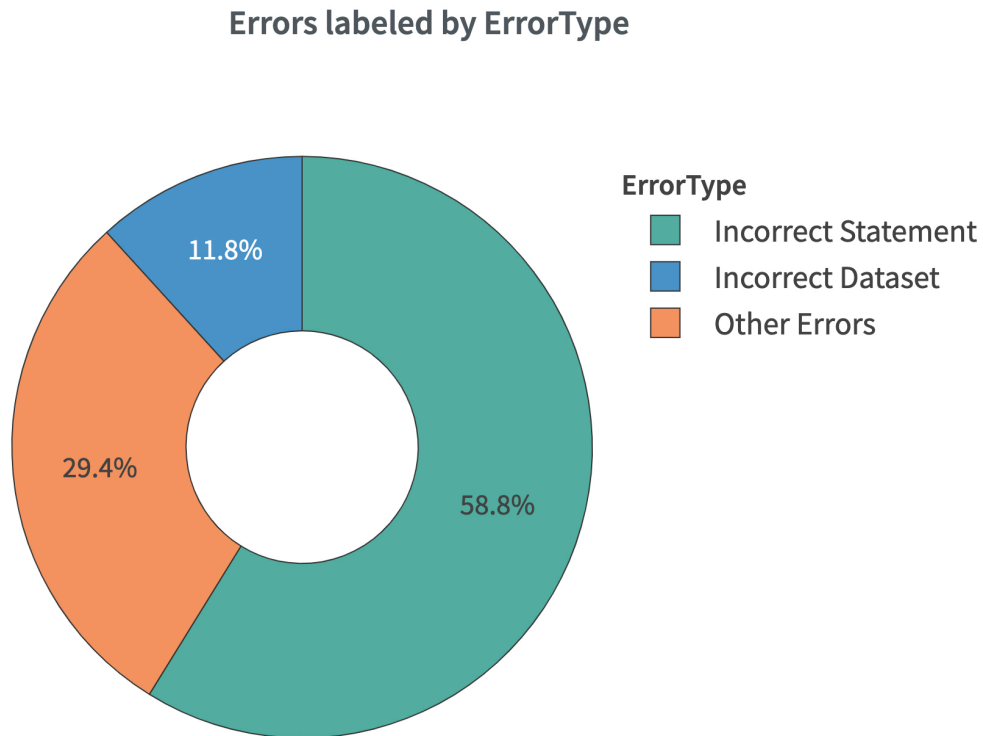


Figure 4.13: Breakdown of the errors made by participants in the EDA task.

We also found that 12% of errors were caused because the user chose the incorrect dataset for their task. Some of the exploratory analysis questions required the user to switch from the currently computed result to the original dataset to get the correct answer. Participants could easily do so in Python as they could store the original dataset in a variable and reference that variable whenever needed. In DataChat, the equivalent to a Python variable is a named dataset. Users can assign a name to the result of a computation in DataChat. DataChat also auto-generates names for datasets that are created as a result of a skill execution. Users can then use

these names to switch to the appropriate dataset for the current task at hand.

Unlike Python, where merely referencing a variable in code would implicitly use the data associated with the variable, in DataChat, the user has to explicitly signal their intention to use a particular dataset for computation before running the actual computation itself. Understanding this notion of explicit dataset referencing is crucial for the users to work in DataChat. We made sure to emphasize the notion of explicit dataset referencing in DataChat in the training video and tutorials. Despite that, we found that a few participants used the incorrect dataset for some of their EDA tasks.

These and other observations indicate that, despite training, users can still make mistakes that impair the accuracy and usability of the DS pipelines that they create. A DS workflow tool such as DataChat must be built with sanity checks and feedback mechanisms to guide users away from such mistakes, and such efforts are part of the DataChat platform. Users are prompted when they likely need to a new dataset and a one-click option is created to allow them to make this switch easily.

4.4.3 Qualitative Feedback

We were also able to obtain qualitative feedback from some of the participants of the study. Every participant who provided feedback had a positive outlook towards DataChat. They strongly felt that DataChat could help them complete data analysis tasks in a much cleaner, faster, and in a reproducible way compared to Python. Participants liked that they could perform complex data manipulations and visualizations with ease in DataChat. Participants also felt that they were able to obtain better quality models quickly due to DataChat's built-in hyperparameter optimization and pipeline functionality. On the flip side, participants felt that they needed more training to become comfortable with the auto-suggest mech-

Positive	Negative
It is straightforward and the learning curve is low. The user can analyze and visualize data in a short time by simply typing English sentences.	Utterances are rigid. Needs more training to become comfortable.
Hyperparameter optimization was very helpful in exploring models.	The hyperparameter configuration (in English) for the machine learning models is difficult to understand in a short time.
With the given time constraints I think I wasn't able to come up with good models on the Python task, whereas DataChat spit up competitive models right away	Customizing the model felt easier in Python, I felt it was more transparent and I had more control on my settings
Loading data, pre-processing data, filtering out principal features and reusing selected features for different models was easier	Complex features like looping over a parameter space and more involved modeling is not possible
Plotting - Much easier and a much better interactive feel	It wasn't completely intuitive all the time, and I think it would be harder to use without any pandas familiarity. But, I am guessing that the intuition would get better over time.

Table 4.1: Qualitative feedback from the participants of the user study

anism and the explicit dataset referencing. They also felt that as the data science workflow became complex, it was hard for them to keep track of any intermediate results they created as a part of the workflow. Further, one participant noted that it was not possible for expert users to take control over the modeling process in terms of specifying parameter grids and manually iterating over different classes of models. We'd like to note that, in DataChat, advanced users can seamlessly switch to a Jupyter notebook to drive the analysis. All intermediate data products created by the user in DataChat are accessible from the Jupyter environment. This aspect is crucial as it allows advanced users to take control of the analytics process, minimizing frustration when the task at hand cannot be completed using DataChat. This functionality was not available to the users of this study.

Snippets from the qualitative feedback provided by users is shown in Table 4.1. We'd like to thank the participants of the study for their valuable feedback and we hope to incorporate some of the feedback in future versions of DataChat.

4.4.4 Other Observations

Since we held out a subset of the labeled data from the participants in the study, we were able to compare the f1-scores of their models on this (unseen) test set. Participants were able to achieve higher f1-scores on the test set using models built with DataChat compared to Python. Figure 4.14 shows the distribution of the f1-scores on the test data for models trained using Python and DataChat.

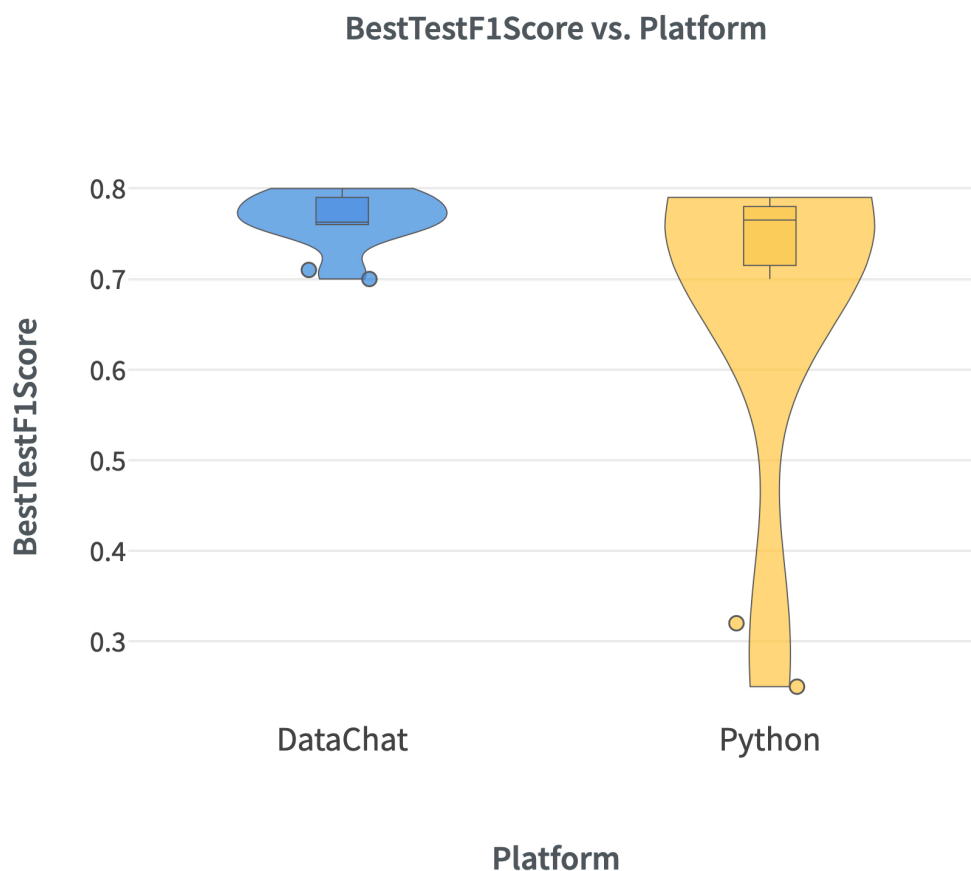
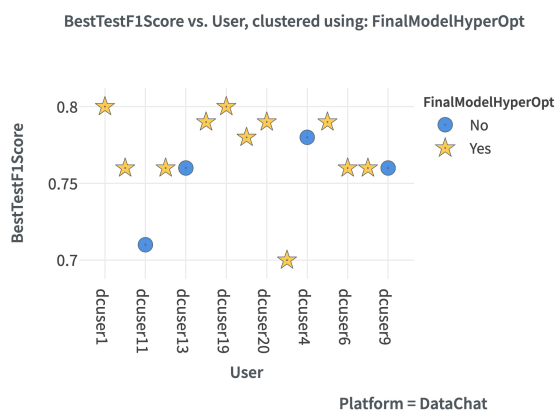


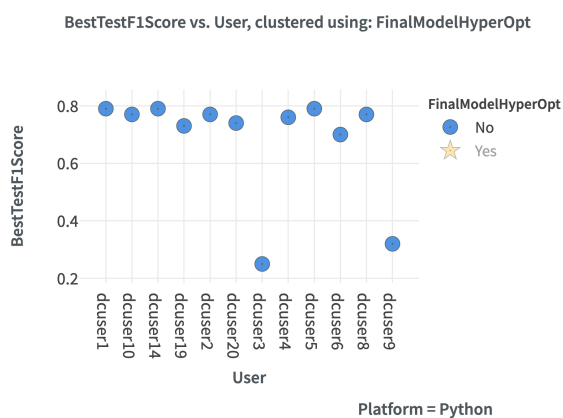
Figure 4.14: Distribution of f1-scores on the test dataset.

Figure 4.15 compares the f1-scores of the top models produced for using Python and DataChat. Four of the top scoring models produced by the participants were trained end-to-end using DataChat's AutoML framework. 11 out of 15 high scoring models built in DataChat used the hyperparameter optimization functionality in the AutoML framework.

Participants were able to consistently train high-quality models in DataChat, with the f1-scores for the models ranging from 0.7 to 0.8, with



(a) F1-scores of best models trained in DataChat for each user



(b) F1-scores of best models trained in Python for each user

Figure 4.15: Comparison of f1-scores of the best model for each user trained using DataChat and Python

a standard deviation of 0.03. Though some users were able to train high quality models using Python (f1-score of 0.79), participants also produced low quality models (f1-score of 0.25). The standard deviation of the f1-scores was 0.18.

Built-in sanity checks We observed that DataChat’s built-in checks

and feedback mechanisms (encoded in DataChat’s ML knowledge base) helped users train better models. Two of the participants were notified by DataChat when they attempted to train a model with the label column as one of the features. Additionally, DataChat’s AutoML framework automatically discarded the less informative user id feature during model training.

Class imbalance handling The dataset used for the study was skewed with respect to the predictor variable, with 70% of the points belonging to one class. However, models were able to obtain high f1-scores without class imbalance handling. The top scoring model (f1-score of 0.8) was trained using DataChat. The pipeline for this model involved hyperparameter optimization, but did not involve handling skew in the predictor variable. Of the models produced in DataChat, 3 models were trained with class imbalance handling. These models achieved a f1-score of 0.76. None of the models trained in Python used class imbalance handling in their training pipeline. Although class imbalance handling wasn’t necessary for this task, one could imagine a scenario where handling skewed class labels becomes critical to training high quality models. In such a scenario, DataChat’s AutoML engine can achieve high quality models by incorporating skew handling as a part of its ML pipeline graph.

4.5 Conclusion

This chapter presents DataChat’s ML Pipeline infrastructure that enables users to build scalable, reusable, framework-agnostic model training pipelines. We also describe DataChat’s AutoML framework that leverages the ML pipeline infrastructure to simplify the model training process. We show that our AutoML framework can incorporate transparency and user feedback into the modeling process by generating model, pipeline, and data explanations.

Our user study results show that non-experts could train high-quality models in DataChat by using DataChat’s CNL utterances, to compose ML pipelines. We also found that users were able to become productive in DataChat with a minimal amount of training. Feedback from the user study has inspired interesting future directions regarding improving DataChat’s user interaction model and machine learning modeling processes.

5

Conclusion and Future Work

Enterprises increasingly employ a wide array of tools and processes to make data-driven decisions. However, there are significant inefficiencies in the enterprise-wide workflow that stem from the fact that business workflows are expressed in Natural Language (NL) but the actual computational workflow has to be manually translated into computational programs.

In this thesis, we present an initial approach to bridge this gap by targeting the DS component of enterprise workflows. In this initial approach, we allow a data scientist to assemble data analytics pipelines using NL via a conversational agent. Our approach uses a restricted subset of NL known as Controlled Natural Language (CNL) to get around the ambiguity in NL. Further, we recognize that DS workflow components are often templated. Putting these two insights together, we develop a practical system called Ava that uses an unambiguous version of NL known as CNL to program DS workflows.

In contrast to some of the other NL specification-to-task translation approaches such as NL-to-SQL and NL-to-viz, we seek to map NL task specifications for a broader set of data science tasks such as data querying, data cleaning, data visualization, and model training to actual computational programs in Python, R, SQL, and Javascript. This broader set of

possible task specifications requires the system to be able to unambiguously identify the task domain, e.g., data querying, in-order to generate the correct code translation for the task. Initial experiments using ML models for identifying the task domain from uncontrolled NL yielded poor results. The ML models were error-prone (due to the ambiguity inherent in NL), produced probabilistic responses and were sensitive to distributional shifts in the training data. Our CNL approach allows translation from NL task specifications to domain-specific task representations (computational programs) in a deterministic manner without relying on ML models.

The language surface of our initial prototype Ava was very limited and its CNL was very inflexible. We improve upon Ava’s CNL to be more flexible and robust to errors while retaining its simplicity. We formalize Ava’s CNL via a pattern language that uses the concept of entities and intent as its building blocks. We also propose a scalable and resource-efficient end-to-end architecture for a NL conversation based system, which is implemented in a commercially deployed system called DataChat.

Finally, we introduce the notion of an ML Pipeline, which is a DAG-based abstract representation of the various stages involved in the ML model training workflow and the flow of information between those stages. The ML pipeline framework lays the foundation for building scalable, shareable, reusable, and framework-agnostic ML model training workflows. We explore the use of the ML pipeline infrastructure to develop DataChat’s AutoML framework.

Our user study results show that non-experts could train high-quality models in DataChat by DataChat’s CNL utterances to compose ML pipelines. We also found that users could become productive in DataChat, with a minimal amount of training.

5.1 Future Work

Our learnings from the initial Ava prototype and the feedback from early users of DataChat have opened up interesting future research and system design avenues. We discuss some of these directions for future work below.

5.1.1 Model Management

Model training in an iterative process. A data scientist will explore many models before arriving at a model suitable for the task at hand. When selecting or eliminating models, the data scientist generates valuable information that informs their future model selection thought process. Such information, when captured in a structured way, can be shared and reused across the organization. This captured knowledge could inform the model selection of other data scientists within the same organization. In an industry setting, where models are retrained periodically as new data comes in, it is crucial to version models and pipelines, and to keep track of metadata such as library versions to enable the data scientists to easily manage the models that they build over time [32, 33]. DataChat’s ML pipeline infrastructure is the first step towards versioned, managed models in DataChat. In addition to versioning and the querying of model metadata, the model management system should also provide tools to detect issues such as model drift.

5.1.2 Developing tools for collaborative analytics

Data science is a collaborative activity involving various stakeholders within an organization. The data analysis process that generates the most valuable insight involves active communication and collaboration between the data scientist and the different stakeholders and domain experts within

an organization. Without the input of the stakeholders and domain experts, the data scientist lacks sufficient business context to solve the task at hand.

Existing data science tools and platforms allow for the results of the data science process (charts, data files, and models) to be shared with the stakeholders but do not allow for active collaboration between the stakeholders during the end-to-end insights generation process. A data science tool that supports collaboration should allow synchronous collaborative human-to-machine communication and human-to-human asynchronous communication based on facts and the data products that are generated during the data analysis process.

5.1.3 Multi-Modal User Interface for Task Specification

As we continue to enhance DataChat's Machine Learning (ML) capabilities, composing a detailed ML task specification using DataChat's CNL may increase the cognitive overload for users and result in decreased productivity due to the increased complexity of DataChat's CNL. Table 5.1 scores the readability of various utterances in DataChat's CNL when invoking DataChat's AutoML functionality.

As the number of configuration options increases, the Flesch readability score [61] decreases. One could easily envision a scenario where a user specifies tens of parameters to DataChat's AutoML for advanced model exploration. Such complex utterances are difficult to compose and interpret. This observation calls for a hybrid UI-CNL based approach wherein the users signify the initial intent to DataChat via DataChat's CNL utterances e.g., Train model. This initial partial intent specification can then be used to trigger a UI to specify values for the entities in DataChat's CNL specification. This approach would involve storing and tracking dialog states and dynamically generating the UI from the current conversation context.

AutoML CNL	Flesch Reading Ease
Analyze Churn	35
Analyze Churn excluding Id	33
Analyze Churn excluding Id weighting with current	30
Analyze Churn excluding Id weighting with weights with feature-pruning	45
hline Analyze Churn excluding Id weighting with weights with feature-pruning, model-optimization	10
Analyze Churn excluding Id weighting with weights with feature-pruning, model-optimization, temporal-slicing	-15
Analyze Churn excluding Id weighting with weights with feature-pruning, model-optimization, temporal-slicing without autobinng	-17
Analyze Churn excluding Id weighting with weights with feature-pruning, model-optimization, temporal-slicing without autobinng, oversampling	-27

Table 5.1: Flesch reading ease scores for DataChat’s AutoML CNL

5.1.4 Uncontrolled NL to SQL (NL2SQL)

Advances in Natural Language Processing (NLP) such as BERT [62] have achieved impressive results in NL to SQL translation tasks. However, these seq2seq (sequence to sequence) [63] models do not achieve high accuracy as they struggle to generalize unseen query structures. Herzig et al. propose to improve the generalizability of seq2seq models by training the model to map from natural language to an intermediate representation that has stronger correspondence with NL, instead of directly translating from NL to SQL [64]. This approach has improved upon the state-of-the-art results on three text-to-SQL tasks. An interesting future direction in this regard would be to examine the use of DataChat’s CNL as an intermediate representation for the seq2seq models in generating SQL from NL.

Bibliography

- [1] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake Vander-Plas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.
- [2] Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software*, 28(1):1–26, 2008.
- [3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.
- [4] Drew Conway. Datascience venn diagram. <http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram>, Accessed On 04/10/2020.
- [5] Thomas C. Redman. Do your data scientists know the ‘why’ behind their work? <https://hbr.org/2019/05/do-your-data-scientists-know-the-why-behind-their-work>, Accessed On 04/10/2020.

- [6] Greg Satell. Most ai projects fail. here's how to make yours successful. <https://www.digitaltonto.com/2018/most-ai-projects-fail-heres-how-to-make-yours-successful/>, Accessed On 04/10/2020.
- [7] Cynthia Matuszek, Evan Herbst, Luke S. Zettlemoyer, and Dieter Fox. Learning to parse natural language commands to a robot control system. In Jaydev P. Desai, Gregory Dudek, Oussama Khatib, and Vijay Kumar, editors, *Experimental Robotics - The 13th International Symposium on Experimental Robotics, ISER 2012, June 18-21, 2012, Québec City, Canada*, volume 88 of *Springer Tracts in Advanced Robotics*, pages 403–415. Springer, 2012.
- [8] Fei Li and H. V. Jagadish. Nalir: an interactive natural language interface for querying relational databases. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 709–712. ACM, 2014.
- [9] Diptikalyan Saha, Avrielia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. Athena: An ontology-driven system for natural language querying over relational data stores. *Proc. VLDB Endow.*, 9(12):1209–1220, August 2016.
- [10] Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. Eviza: A natural language interface for visual analysis. In Jun Rekimoto, Takeo Igarashi, Jacob O. Wobbrock, and Daniel Avrahami, editors, *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*, pages 365–377. ACM, 2016.
- [11] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020. <https://d2l.ai>.
- [12] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.
- [13] Joseph Weizenbaum. Eliza – a computer program for the study of natural language communication between man and machine. *CACM*, 9(1):36–45, January 1966.

- [14] Kenneth Mark Colby. *Artificial Paranoia: A Computer Simulation of Paranoid Processes*. Elsevier Science Inc., New York, NY, USA, 1975.
- [15] Richard S. Wallace. *The Anatomy of A.L.I.C.E.*, pages 181–210. Springer Netherlands, Dordrecht, 2009.
- [16] Loebner prize. https://en.wikipedia.org/wiki/Loebner_Prize, Accessed On 04/10/2020.
- [17] Jason Williams, Eslam Kamal, Mikhtar Ashour, Hani Amr, Jessica Miller, and Geoffrey Zweig. Fast and easy language understanding for dialog systems with microsoft language understanding intelligent service (luis). In *Proceedings of 2015 SIGDIAL Conference*, 2015.
- [18] Watson conversation. <https://www.ibm.com/cloud/watson-assistant/>, Accessed on 04/10/2020.
- [19] Amazon lex. <https://aws.amazon.com/lex/>, Accessed on 04/10/2020.
- [20] Richard Wallace. The elements of aiml style. *Alice AI Foundation*, 2003.
- [21] Tobias Kuhn. A survey and classification of controlled natural languages. *Comput. Linguist.*, 40(1):121–170, March 2014.
- [22] Dagi Troegner. Grammar for natural language recognition: Adaptation to air traffic phraseology. 2011.
- [23] Rolf Schwitter. Controlled natural languages for knowledge representation. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters (COLING)*, pages 1113–1121, 2010.
- [24] Norbert E. Fuchs, Stefan Höfler, Kaarel Kaljurand, Fabio Rinaldi, and Gerold Schneider. Attempto controlled english: A knowledge representation language readable by humans and machines. In *Proceedings of the First International Conference on Reasoning Web*, pages 213–250, 2005.
- [25] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

- [26] Adam Blum. Microsoft english query 7.5: Automatic extraction of semantics from relational databases and olap cubes. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 247–248, 1999.
- [27] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces (IUI)*, pages 149–157, 2003.
- [28] Yunyao Li, Huahai Yang, and H. V. Jagadish. Nalix: An interactive natural language interface for querying xml. In *Proceedings of the 2005 ACM International Conference on Management of Data (SIGMOD)*, pages 900–902, 2005.
- [29] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, September 2014.
- [30] D. Küpper, M. Storbel, and D. Rösner. Nauda: A cooperative natural language interface to relational databases. *SIGMOD Rec.*, 22(2):529–533, June 1993.
- [31] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. Making the case for query-by-voice with echoquery. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 2129–2132, 2016.
- [32] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec.*, 44(4):17–22, May 2016.
- [33] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: A system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA)*, pages 14:1–14:3, 2016.

- [34] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. Mlbase: A distributed machine-learning system. In *Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.
- [35] Fernando Perez and Brian E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science and Engg.*, 9(3):21–29, May 2007.
- [36] Jpype - java to python integration. <http://jpype.sourceforge.net/>, Accessed on 04/10/2020.
- [37] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.
- [38] Springleaf marketing response. <https://www.kaggle.com/c/springleaf-marketing-response/>, Accessed On 04/10/2020.
- [39] Customer churn prediction. <https://www.knime.org/knime-applications/churn-prediction>, Accessed on 04/10/2020.
- [40] Churn analysis using ava. <http://go.wisc.edu/78415o>, Accessed on 04/10/2020.
- [41] Churn analysis using python. <http://go.wisc.edu/h2ex60>, Accessed on 04/10/2020.
- [42] J. W. Tukey. *Exploratory Data Analysis*. Behavioral Science: Quantitative Methods. Addison-Wesley, Reading, Mass., 1977.
- [43] Functor. <https://www.thoughtco.com/function-word-grammar-1690876>, Accessed On 04/10/2020.
- [44] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.
- [45] Microservices. <https://microservices.io/>, Accessed On 04/10/2020.
- [46] Reactjs. <https://reactjs.org/>, Accessed On 04/10/2020.

- [47] Redis. <https://redis.io/>, Accessed On 04/10/2020.
- [48] Rabbitmq. <https://www.rabbitmq.com/>, Accessed On 04/10/2020.
- [49] Docker compose. <https://docs.docker.com/compose/>, Accessed On 04/10/2020.
- [50] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.
- [51] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, pages 785–794, New York, NY, USA, 2016. ACM.
- [52] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [53] Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Drogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6639–6649, 2018.
- [54] Kaggle 2020 survey. <https://www.kaggle.com/c/kaggle-survey-2020>, Accessed On 05/02/2021.
- [55] Guillermo Navas-Palencia. Optimal binning: mathematical programming formulation. [abs/2001.08025](https://arxiv.org/abs/2001.08025), 2020.
- [56] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16:321–357, 2002.

- [57] Saleema Amershi, Maya Cakmak, William Bradley Knox, and Todd Kulesza. Power to the people: The role of humans in interactive machine learning. *AI Magazine*, 35(4):105–120, Dec. 2014.
- [58] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.
- [59] Telco churn analysis. <https://www.kaggle.com/c/churn-analysis>, Accessed On 05/02/2021.
- [60] Kerry Rodden, Hilary Hutchinson, and Xin Fu. Measuring the user experience on a large scale: User-centered metrics for web applications. In *Proceedings of CHI 2010*, 2010.
- [61] John R. Talburt. The flesch index: An easily programmable readability analysis algorithm. In Diana Patterson, editor, *Proceedings of the Fourth International Conference on Systems Documentation, SIGDOC 1985, Ithaca, NY, USA, June 18-21, 1985*, pages 114–122. ACM, 1985.
- [62] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.
- [63] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

- [64] Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. Unlocking compositional generalization in pre-trained models using intermediate representations. *CoRR*, abs/2104.07478, 2021.