Learning to Map Natural Language to General Purpose Source Code

A dissertation

submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Luke S. Zettlemoyer, Co-Chair

Alvin Cheung, Co-Chair

Yejin Choi

Program Authorized to Offer Degree:

Computer Science and Engineering

University of Washington

Abstract

Learning to Map Natural Language to General Purpose Source Code

Srinivasan Iyer

Co-Chair of the Supervisory Committee:

Associate Professor Luke Zettlemoyer

Assistant Professor Alvin Cheung

Computer Science and Engineering

Models that automatically map natural language (NL) to source code in general purpose languages such as Java, Python, and SQL find utility amongst two main audiences viz. developers and non-expert users. For developers, they enable use-cases such as functioning as a NL assistant in programming IDEs, verifying the consistency of code documentation with code changes, and answering "how to" questions, for developers using new languages. For non-expert users, they enable use-cases of being able to communicate with databases, devices and applications, or of visualizing data, without having to learn to write computer programs.

Developing these models is challenging because of contextual dependencies of the target code, the lack of alignment between NL and code tokens, syntactic and semantic requirements of the target code, and the prohibitively expensive cost of annotating training data. Furthermore, whilst developers can see and manipulate the generated code, non-expert users only see the output of execution, and therefore have the additional constraint of the generated code being exactly correct and executable. Finally, for users to trust models that automatically produce code, particularly in high-cost scenarios, it is important for models to provide an explanation of the generated code back to the user.

This dissertation presents tasks, training methods/resources and new models for mapping NL to source code for both developers and non-expert users, and is divided into four parts. In the first part, we formalize the task of contextual code generation from NL for developers. We present ways to obtain inexpensive training datasets from large online code repositories, followed by methods to incorporate contextual awareness

into syntax-guided neural models to improve performance on the task.

The second part shifts focus from developers to non-expert users, where we present methods to build NL interfaces that allow non-expert users to query databases by automatically mapping their NL requests to database SQL queries. Our methods are geared towards building deep learning models that improve in performance over time by leveraging user feedback and annotations obtained from crowd programmers, and open up inexpensive ways to build accurate NL interfaces for arbitrary database schemas.

The third part of this dissertation presents the use of programmatic idioms as a means to significantly improve training time, as well as performance on both the NL to code tasks of parts 1 and 2. We discuss algorithms to extract frequently used programmatic idioms and train neural models to learn to apply them during code generation.

Finally, we present models that describe the functionality of source code to users in NL as a first step towards building trustworthy language to code systems. Overall, this dissertation presents efficient deep learning models and training paradigms to map language to general purpose source code that will enable numerous applications for non-expert users as well as developers.

Acknowledgements

My PhD would not have been possible if it weren't for the mentorship, support, and friendship of a number of people, and I would like to express my heartfelt gratitude to each and every one of them. I don't aim to name everyone here, since any such attempt by me would be futile.

I would like to profusely thank my advisors, Luke Zettlemoyer and Alvin Cheung for their mentorship and guidance throughout my PhD, and for giving me the freedom to pursue my own research directions. Their encouragement and flexibility has transformed my PhD experience from what is typically a very stressful time for most students, into a stress-free and pleasant experience. I've been incredibly lucky to have had Luke as my advisor, from whom I've learnt so much, not just about being a creative and independent researcher, but also about exhibiting modesty and generosity at all times. I would also like to thank my thesis committee members, Yejin Choi and Amy Ko, for taking the time to provide feedback and suggestions about my research agenda.

I am incredibly and forever grateful to Yannis Konstas, who worked very closely with me during my initial PhD years to get me off the ground, showed me the ropes to becoming a successful researcher, and played a significant role in teaching me to write publications. I would also like to plentifully thank my Master's advisor, Nigam Shah, for introducing me to scientific research, for encouraging me to pursue a career in research, and for enabling me to be successful. I am heavily indebted to Paea LePendu for patiently teaching me research methodology during my initial years, for the exciting code hacking sessions, and for showering me with encouragement to push myself to set and achieve higher goals. I'm also thankful to Anna Bauer-Mehren and Rave Harpaz for endless research discussions and initial encouragement.

I've had the good fortune of being able to collaborate with some of the best people in my field, who have broadened my horizons, and in the process, have also become very close to me. This includes Alane

Suhr, Antoine Bosselut, Mark Yatskar, Yoav Artzi, Victor Zhong, Hannah Rashkin, Pradeep Dasigi, Rajas Agashe, Matt Gardner, Urvashi Khandelwal, Asli Celikyilmaz, Thomas Wolf, and Marjan Ghazvininejad. I'm beholden to my internship mentors, Jayant Krishnamurthy, as well as Mike Lewis, who always pushed me to think big and to take risks in order to have large impact.

At various stages during my PhD, I've received excellent advice and mentorship, particularly from Omer Levy, Dallas Card, Gabriel Satanovsky, Aditya Vashistha, Kenton Lee, Rik Koncel-Kedziorski, Marco Tulio Ribeiro, Bill Howe, Dan Halperin, and Magdalena Balazinska. I'm also immensely thankful to my friends from UWNLP and the broader NLP community for insightful research discussions, for reading paper drafts and listening to practice talks, for celebrating successes with me and hearing me whine about rejections. Some of these folks include: Julian Michael, Luheng He, Ofir Press, Terra Blevins, Tim Dettmers, Jesse Thomason, Sewon Min, Christopher Clark, Kelvin Luu, Elizabeth Clark, Maarten Sap, Yonatan Bisk, Swabha Swayamdipta, David Golub, Victoria Lin, Nicholas Fitzgerald, and Siva Reddy. I've also received an incredible amount of support from the UW Databases Group - particularly from Jenny Ortiz, Brandon Haynes, Dominik Moritz, Laurel Orr, Chenglong Wang, Shrainik Jain, and Shana Hutchison, and have very fond memories of brunches and social gatherings.

Hiking the beautiful mountains of the Pacific Northwest has been my solace in times of hardship and stress, as well as a cherished celebratory ritual throughout my PhD. I'm incredibly thankful to my close friends who have gone on countless hiking adventures with me: Naman Gupta, Yuguang Lee, Angli Liu, Maaz Ahmad, and Kshitij Kumar.

I was only able to survive grad school owing to the continued and dependable support of my close friends with whom I've shared laughs, joys and sorrows over countless coffee chats, meals, and late night drinks. These include Guna Prasaad, Spandana Gella, Sachin Mehta, Koosha Khalvati, Maaz Ahmad, Deepali Aneja, Mandar Joshi, Gagan Bansal, Eunsol Choi, Chandrakana Nandi, Cong Yan, Aravind Rajeshwaran, Rahul Kidambi, Maliha Mustafa, Eunice Jun, Krishna Pilutla, Shumo Chu, Svetoslav Kolev, Aman Kalia, Swati Padmanabhan, Vidur Joshi, Aman Nijhawan, and Anant Bhardwaj. I'm also grateful to Siena Dumas, Dan Radion, and Gaurav Mukherjee for their help and support along the way, and to Ram Srinivasan for heavily influencing me to undertake my PhD.

A large part of my smooth PhD experience was owing to Elise deGoede Dorough, who has been of

immeasurable help for everything from RAships to making sure I met all my requirements on time, and Chiemi Yamaoka, who took excellent care of all our group's needs from conference travel to GPUs.

I'm incredibly grateful to my friends from my undergraduate institution, particularly: Ashish Kurmi, to whom I owe an eternal debt of gratitude for pushing me to discover what I'm capable of and to reach for the stars; Aliasgar Arsiwalla, for encouraging me to take risks and to not get comfortable; and Madhur Auti, for being a dependable pillar of friendship and comfort.

Lastly and most importantly, I owe all my successes to my late parents: My father, Dr. S.V. Krishnan, who always encouraged me to strive for excellence in all areas of life, who absolutely enjoyed the pursuit of knowledge, and who inspired me to pursue my PhD; and my mother, Vijayalakshmi, who has laid out an exceptional standard for kindness, generosity and compassion towards everyone, that I strive to achieve everyday. This dissertation is dedicated to them and I'm positive that they would both be incredibly proud. I'm also extremely grateful to my brother, Laxmiprasad Iyer, who played a vital role in getting me excited about computer science research, and my cousin, Lakshmi Vaz, who has showered me with affection, support and encouragement at all times.

To my parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mapping instructions in Natural Language (NL) into programs is a popular task in NLP with numerous applications for non-expert users – such as voice assistants, home automation devices, and NL interfaces to databases (NLIDBs), as well as for developers – such as automatic code generation in IDEs, bug-identification, and code retrieval from large corpora. It is particularly more exciting and relevant today, when software is ubiquitous and the number of people writing source code is growing at a striking rate. A monumental number of software packages are being released every month, making it impossible for programmers to adequately familiarize themselves with most APIs. Training machine learning models to automatically generate code from language can significantly improve developer efficiency and source code quality, and also help non-expert users easily communicate complex intents with devices using natural language.

Earlier NL to code systems focused mainly on the task of semantic parsing by mapping NL to special-purpose intermediate meaning representations such as lambda calculus [Zettlemoyer and Collins, 2005], which worked well within research communities but was considered too expensive and inextensible for real-world deployment. In this dissertation, we develop deep learning methods for the task of mapping NL to source code in general purpose programming languages such as Java, Python, and SQL, which can be used for multiple applications in multiple domains. Mapping to these languages also allows us to easily and regularly solicit inexpensive dataset annotations by leveraging the expertise of their large programming communities online, which in turn enables periodic retraining of models and therefore, model extensibility. Similarly, mapping NL directly to general purpose languages used by developers, as opposed to via inter-

mediate representations, allows us to inexpensively use well documented large scale online source-code corpora to train models.

Recently, neural attentional encoder-decoder models have been shown to achieve very high performance on various sequence to sequence tasks [Bahdanau et al., 2015; Kiddon et al., 2016; Rush et al., 2015]. However, NL to code tasks have additional challenges requiring significant architectural changes to these models. Unlike sequence transduction tasks such as machine translation, NL to code tasks involve structured prediction without any definite alignments between source NL words and the target code tokens. Under this setup, a small change amongst just a few words in the NL can result in quite a significant change in the target code. For example, when generating SQL, changing the phrase *find the largest* to the phrase *find the second largest* means that the target code requires an entire extra subquery to obtain the desired result. NL to code models also have the added constraint of requiring the output code to respects the syntactic and semantic constraints imposed by the target programming language, and thus, must be aware of the target parsing grammar. Furthermore, while general structured prediction models are typically autoregressive at the token level, NL to code models specifically can benefit from decoding source code in terms of higher level code structures, known as programmatic idioms [Allamanis and Sutton, 2014]. This is similar to the approach taken by human programmers to develop source code and can improve both the training speed and accuracy of code generation models.

Obtaining labeled datasets for training NL to code models requires annotation by skilled programmers and is thus, prohibitively expensive. When available, this supervision is limited to obtaining only (NL, code) tuples without the derivation process to get from the source to the target Zettlemoyer and Collins [2005], leading to the derivation process being typically modelled as being latent. This also necessitates researchers to seek out ways to use weak supervision [Krishnamurthy and Mitchell, 2012; Artzi and Zettlemoyer, 2013] or obtain distant supervision for model training. The significant annotation cost also prohibits models from being retrained periodically on newly annotated data to adapt to changing NL distributions when deployed, and results in models that are trained and deployed only once.

In addition to depending on the input NL instruction, the target source code is also heavily influenced by context, be it existing source code in the case of developers using programming IDEs [Iyer et al., 2018] , or prior NL interactions in the case of non-expert users [Zettlemoyer and Collins, 2009; Suhr et al., 2018].

Changes in programming context can significantly change the desired target output. For example, the context can specify the set of packages and APIs, or predefined variables and methods that should be used in the target code, or can also resolve various kinds of ambiguity that can result by considering the NL instruction alone without the context (see Section 1.2). Successful NL to code systems must explicitly model the context under which instructions are situated.

Evaluating the generated output of NL to code models is challenging, particularly when the target code is general purpose, since there are often multiple code snippets that can achieve the same result. A combination of different evaluation metrics such as exact match Ling et al. [2016], bleu score Papineni et al. [2002]; Ling et al. [2016] and execution accuracy are typically used to evaluate generated code under different circumstances. For example, when the intended audience are developers, we would like a metric that can also assign partial credit to model outputs, since under this scenario, developers can understand the output and make quick manipulations to fix minor errors in generation. Since developers would likely use NL to code systems to build smaller pieces of a larger incomplete project, it is typically impossible to train models using rewards based on execution of the generated source code. On the other hand, when generating SQL queries for NLIDBs used by non-expert users who do not understand the SQL output, it is important for the results to be exactly correct. In this case, metrics such as exact match, and also execution accuracy are more appropriate, since SQL queries are typically standalone. Particularly for non-expert users, it is also essential to train code to NL models, that explain generated source code back to users in NL, in order to ensure that users aren't misled or deceived by any model failures.

In the first two chapters of this dissertation, we present deep learning models that address the contextual NL to code task for developers, as well as NL-based querying tasks for non-expert users, focusing on the distinctive challenges that arise for each of these audiences. We present solutions that take a holistic view of these systems, by addressing issues of data gathering, model efficiency and scalability, as well as model adaptation to new domains and model extensibility over time. After an in-depth presentation of models for these audiences, we discuss general novel methods to scale up NL to code models to larger datasets, while simultaneously improving performance. Finally, we also present models to describe source code in NL, which is invaluable in establishing trust between the model output and its users. In the process, we also release three new datasets containing tuples of (NL, general purpose source code) for the tasks of

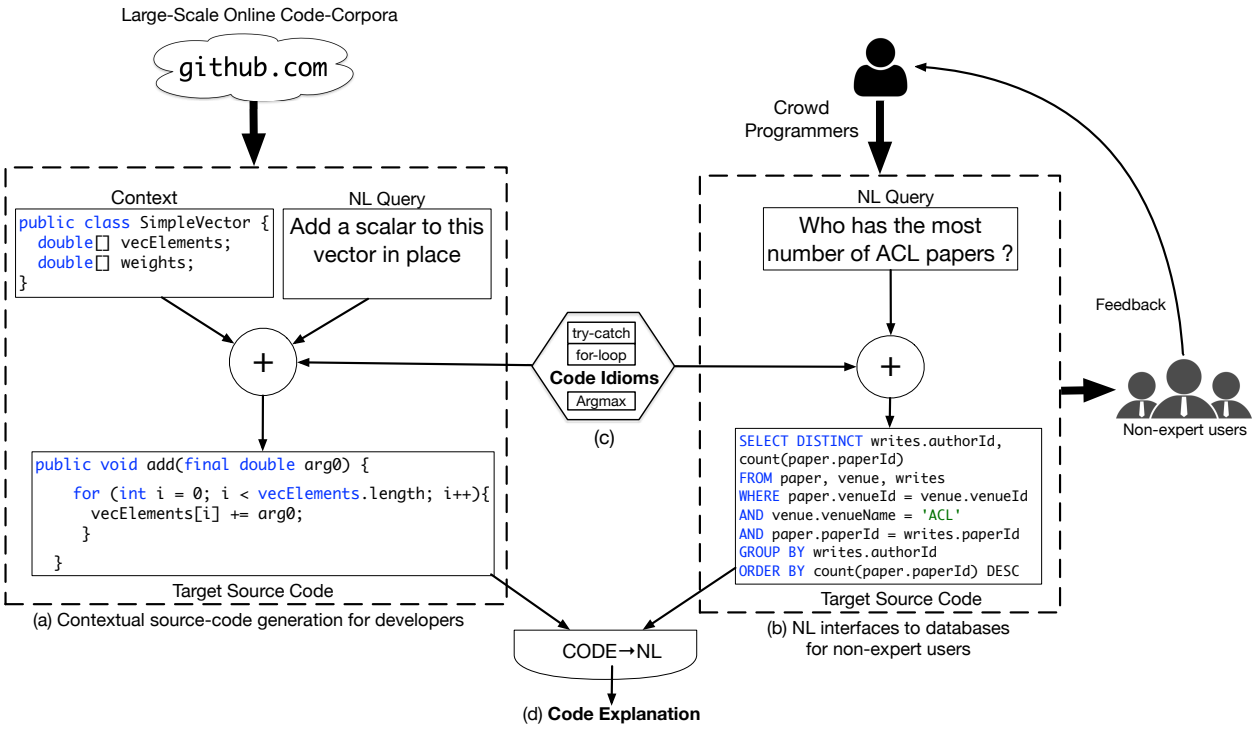

Figure 1.1: This figure presents a holistic view of the various aspects of mapping NL to general purpose source code that we discuss throughout this dissertation. (a) We present deep learning models that address the contextual NL to code task for developers by training on datasets leveraged from large-scale online code corpora, (b) as well as extensible database querying models for non-expert users by using a combination of user feedback and crowd programmers for obtaining supervision. (c) Following this, we discuss general novel methods to scale up NL to code models to larger datasets, while simultaneously improving performance, using code idioms. (d) Finally, we also present models to describe source code in NL, which is invaluable in establishing trust between the model output and its users.

contextual code generation for developers (CONCODE), for mapping NL to SQL queries for use by non-expert users (SCHOLAR), and for the task of generating functional descriptions of source code (CODENN). Figure 1.1 presents a big picture overview of how all these components fit together. Overall, this dissertation takes important steps towards the development of inexpensive ways to train efficient, robust and extensible machine learning based systems for mapping NL to general purpose source code.

In the remainder of this section, we will first present a brief evolutionary history of tasks, methods and datasets relating to mapping NL to programmatic representations in Section 1.1, followed by a deeper dive into some of the challenges that we mentioned above and ways we address those challenges in this dissertation in Section 1.2. Finally, we present an outline of the chapters in this dissertation.

1.1 Background

While our dissertation focuses on the broad task of mapping natural language to general purpose source code for developers as well as non-expert users, until a few years ago, this task has primarily been studied in the research community as a semantic parsing task i.e. mapping NL to executable meaning representations with use-cases catering to non-expert users. In this section, we first present a brief history of semantic parsing systems from the rule based systems of the 1960s upto the neural models in use today, including an evolution of tasks, datasets and modeling paradigms. Following this, we discuss the more recent history of generating general purpose source code for developers in programming IDEs.

1.1.1 Early Systems

Mapping NL to executable programs has received a significant amount of research interest since as early as the 1960s. Early research focused primarily on translating NL instructions into queries that can be executed against a target database. A notable example is the LUNAR system [Woods, 1972] which allowed users to use NL to query a database containing the chemical analyses of moon rocks. These systems were extended to maintain a dialogue with the user permitting the user to clarify and refine the NL instruction (for example, RENDEZVOUS; [Codd, 1974]). These systems made heavy use of hand-engineered rules that mapped patterns in the NL or elements in its syntax parse tree to various query constructs and could only work on a limited subset of natural language with a restricted NL syntax. This also resulted in these systems being incredibly difficult to port to new domains. A number of these systems mapped the NL into an intermediate meaning representation which is database independent, and employed a translation module to convert this into an executable query. An excellent summary of these early systems is presented in Androutsopoulos et al. [1995].

1.1.2 Statistical Semantic Parsers

Statistical methods to map NL to intermediate logical forms gained popularity in the last two decades. These systems did away with the various hand crafted components used in the past and used corpora of NL paired with their queries to induce parsers. The CHILL system [Zelle and Mooney, 1996] automatically learned rules using techniques from Inductive Logic Programming to control the actions of a shift reduce parser, in

order to map NL to a meaning representation based on logic grammars. This was applied to a database of US Geography facts (GEO880) and its performance was improved by Tang and Mooney [2001] using multiple clause constructors from multiple learners. These systems assumed a fixed lexicon, and Thompson et al. [1999] later developed the WOLFIE system to automatically learn a lexicon for CHILL. Kate and Mooney [2006] trained SVM classifiers based on string subsequence kernels for each production in the MRL and then used them to compositionally build complete MRs. Zettlemoyer and Collins [2005] learn to induce a probabilistic Combinatory Categorial Grammar (CCG) from corpora comprising NL questions annotated with lambda calculus meaning representations.

A popular corpus for training semantic parsers was a set of queries related to flight travel issued to an Air Travel Information System (ATIS) database, which also included dialogues between users and the system, and several previous learning methods [Papineni et al., 1997; Miller et al., 1996; Ramaswamy and Kleindienst, 2000; He and Young, 2006] have been applied to this domain. The following is an example of a dialog from ATIS comprising questions posed by a user paired with lambda calculus queries to the database.

(a) show me the flights from boston to philly

$\lambda x.\text{flight}(x) \wedge \text{from}(x,\text{bos}) \wedge \text{to}(x,\text{phi})$

(b) show me the ones that leave in the morning

$\lambda x.\text{flight}(x) \wedge \text{from}(x,\text{bos}) \wedge \text{to}(x,\text{phi}) \wedge \text{during}(x,\text{morn})$

(c) what kind of plane is used on these flights

$\lambda y.\exists x.\text{flight}(x) \wedge \text{from}(x,\text{bos}) \wedge \text{to}(x,\text{phi}) \wedge \text{during}(x,\text{morn}) \wedge \text{aircraft}(x) = y$

Zettlemoyer and Collins [2009] extended their approach using CCGs to parse these sentences in a context dependent fashion by maintaining explicit, lambda-calculus representations of salient discourse entities and using a context-dependent analysis pipeline to recover logical forms. Kwiatkowski et al. [2010] generalize this approach to work with a more diverse set of natural languages and meaning representations by using higher-order unification to define a hypothesis space containing all grammars consistent with the training data, and developing an online learning algorithm to efficiently search this space while simultaneously estimating the parameters of a log-linear parsing model. Finally, Semantic Parsing has also been viewed as a machine translation (MT) task [Andreas et al., 2013; Papineni et al., 1997; Wong and Mooney, 2006]

(revisited later using neural methods; see Section 1.1.4).

1.1.3 Weak Supervision

Owing to the significant cost of obtaining annotated logical forms, researchers explored means to learn from easier forms of supervision. Clarke et al. [2010] learn semantic parsers from training data comprised of questions paired with database answers without the need for annotated logical forms. Liang et al. [2013] take a similar approach where they introduce a new meaning representation (λ-DCS) but model it in a latent fashion to not require annotations. Artzi and Zettlemoyer [2011] extend this approach to learning parameters of a CCG grammar in a latent fashion from conversational logs.

Artzi and Zettlemoyer [2013] present an approach that learns a joint model of meaning and context for executing NL instructions using weak supervision provided by the end state validation i.e. a reinforcement learning style approach. Chen and Mooney [2011] take similar approaches of learning through system demonstrations. Goldwasser and Roth [2014] use binary user-feedback after interpreting the generated logical forms as the only means of supervision for learning Semantic Parsers, a strategy we will revisit later. Nevertheless, Yih et al. [2016] demonstrated that learning from labeled queries is significantly more effective than learning from answers and one can obtain semantic parsers with high accuracies using MRs at a cost comparable or lower than one built by obtaining just answers.

Porting semantic parsers to new domains has always been a challenge, from the early days of using semantic grammars extending even to the statistical approaches that relied on annotations. To test the ability of semantic parsing algorithms to generalize, researchers trained parsers for answering questions on large knowledge bases with hundreds of domains such as FREEBASE [Bollacker et al., 2008]. Strategies to achieve this have included distant supervision [Krishnamurthy and Mitchell, 2012], schema matching [Cai and Yates, 2013], using alignments with external web text and knowledge bases [Berant et al., 2013], graph matching [Reddy et al., 2014], using paraphrasing methods [Berant and Liang, 2014] to exploit large amounts of text not covered by the knowledge base, and using on-the-fly ontological matching [Kwiatkowski et al., 2013]. However, Berant et al. [2014] showed that simple information extraction systems could perform at par with state of the art semantic parsers, thus demonstrating that the compositionality of the required logical forms to answer FREEBASE questions was very limited and much simpler than general purpose database

queries. To address this issue and simultaneously increase both the breadth of the knowledge source and the depth of logical compositionality, Pasupat and Liang [2015] proposed a new task of answering a question using an HTML table as the knowledge source and released WikiTableQuestions; a dataset of questions posed by Amazon Mechanical Turk (AMT) workers on 2,108 Wikipedia tables. Liang [2016] provides a brief history of the state of Semantic Parsing from early systems upto this stage, also describing the salient components of these systems and highlighting key challenges in building them.

1.1.4 Neural Semantic Parsing

Recent research has demonstrated that neural encoder-decoder models with attention are effective for tasks such as machine translation [Bahdanau et al., 2015] and natural language generation [Kiddon et al., 2016]. Neural models have also achieved near state of the art results in semantic parsing tasks to map NL to lambda calculus expressions [Dong and Lapata, 2016; Jia and Liang, 2016] without the need for any manual feature engineering. Instead of directly generating a sequence of code tokens, recent methods focus on constrained decoding mechanisms to generate syntactically correct output using a decoder that has a dynamically-determined modular structure paralleling the structure of the abstract syntax tree (AST) of the code [Rabinovich et al., 2017; Krishnamurthy et al., 2017; Yin and Neubig, 2017]. In Chapter 4 we equip neural decoders with the ability to decode in terms of higher level code structures known as programmatic idioms, which further improves accuracy for general purpose source code generation and enables neural models to scale up to larger datasets. Many of the task/method variations that were explored in the pre-neural era were revisited under the neural paradigm; for example, Suhr et al. [2018] present a neural approach for the task of mapping NL to SQL queries that are context dependent, on the ATIS domain, by copying snippets from previously generated SQL queries.

1.1.5 General Purpose Semantic Parsing

Most of the earlier semantic parsing methods that use intermediate meaning representations have seen very limited use in practice since they are not supported by standard database implementations. One of our goals is to learn models to perform Semantic Parsing directly to executable programs in general purpose languages such as full SQL and Java and bypass intermediate meaning representations. A few systems have

been developed to directly generate SQL queries from NL [Popescu et al., 2003; Giordani and Moschitti, 2012]. However, all of these systems make strong assumptions on the structure of queries: they use manually engineered rules that can only generate a subset of SQL, require lexical matches between question tokens and table/column names, or require questions to have a certain syntactic structure. Poon [2013] introduce an unsupervised method (GUSP) to map NL to an intermediate representation that is then deterministically converted to SQL. The mapping is done by annotating the dependency-tree nodes and edges with latent states, and learning a probabilistic grammar using EM while leveraging the database for indirect supervision. However, it requires a lot of SQL specific engineering (for example, special nodes for argmax) and is hard to extend to more complex SQL queries. Recently, Zhong et al. [2017] released a large dataset of NL queries on Wikipedia tables (WikiSQL), paired with SQL queries that answer them and various neural approaches using slot filling on sketches [Xu et al., 2017], entity typing [Yu et al., 2018a] and multi-channel pointer networks [Sun et al., 2018]. The complexity of the SQL queries in this dataset was limited since all queries could be described using a handful of query templates. In Iyer et al. [2017] (see Chapter 3) we demonstrate that supervised neural encoder decoder models are powerful enough to learn to map NL instructions directly to arbitrary SQL queries and that it is cheap to obtain annotations of SQL for NL instructions by employing crowd programmers as well as using data augmentation methods.

Recently there has been focus on generating programs/representations other than SQL, such as regular expressions [Kushman and Barzilay, 2013; Locascio et al., 2016], short if-this-then-that (IFTTT) recipes [Quirk et al., 2015], a system of equations [Kushman et al., 2014; Roy et al., 2016], map queries [Haas and Riezler, 2016], a sequence of API calls [Gu et al., 2016b], and Abstract Meaning Representation (AMR) [Artzi et al., 2015; Misra and Artzi, 2016]. Gulwani and Marron [2014] use program synthesis to generate a subset of spreadsheet data analysis manipulation programs from NL using a specially designed DSL, which they publish as an Excel add-in. Lin et al. [2018] release a dataset collected from the web that maps NL instructions to bash command line programs and train a template-level neural sequence model on it. Researchers have found that neural models are capable of directly generating code in general purpose programming languages and one advantage of this is that labels/supervision can be obtained by non-linguist crowd programmers who are much easier to train and employ.

Ling et al. [2016] generate Java and Python source code from NL for card games conditioned on categor-

ical card attributes. These programs, however, are highly templated and represent several implementations of the same Java interface. There is also work on generating single lines of Python code from manually labeled per-line comments [Oda et al., 2015]. In Chapter 2 of this work, we introduce a new task of generating general purpose Java programs from NL based on the environment in which the code resides, following the frequently occurring pattern in large repositories where the code depends on the types and availability of variables and methods in the environment. We also introduce a new dataset (CONCODE) that contains a larger variety of output code templates than existing datasets built for a specific domain, and is the first to condition on the environment of the output code. Also, by design, it contains examples from several domains, thus introducing open-domain challenges of new identifiers during test time. Subsequently, Agashe et al. [2019] study the task of code generation under the paradigm of interactive programming, where users write exploratory programs in blocks comprising NL and code (for example, in Jupyter notebooks). Here, the target source code is dependent on a sequence of prior NL and source code interactions.

1.1.6 Deploying Semantic Parsers

Building a semantic parser for a new domain has always been very cumbersome since it requires obtaining potential NL instructions, annotations of meaning representations, feature engineering for training models and also writing components to translate the MRs to executable code. Owing to this, their use has predominantly been restricted to within research communities. Wang et al. [2015] demonstrate that its possible to train a semantic parser for a new domain in a few hours by using a simple grammar to generate logical forms paired with canonical utterances, which can then be paraphrased to natural utterances using crowdsourcing and subsequently used as training data. In Chapter 3, we demonstrate that by training models that directly map NL to SQL queries, we can hire online SQL programmers to provide real-time supervision and thus, deploy a model that improves over time. We show that we could achieve an accuracy of 63% for an NLIDB for an academic database within three days of usage. Wang et al. [2017] demonstrate an approach where they allow users to “naturalize" the programming language incrementally by defining alternative, more natural syntax and increasingly complex concepts in terms of compositions of simpler ones, an idea similar to the early ASK system [Thompson and Thompson, 1983, 1985]. Over the course of three days, these users went from using only the core language to using the naturalized language in 85.9% of the last 10K utterances.

Another important line of research is how to handle situations when Semantic Parsers generate the wrong program. This can thoroughly mislead the user, particularly in NLIDBs where only the output is presented to the user. The PRECISE system by Popescu et al. [2003] addressed this issue by building a system with 100% precision that would only attempt to generate results for semantically tractable queries and guaranteed that the output was correct. Early systems [Mueckstein, 1983; Damerau, 1985; De Roeck and Lowden, 1986; Koutrika et al., 2010; Ngonga Ngomo et al., 2013] used various rules to translate a generated database query back into an English description, which was then presented to the user to verify, sometimes with various graphical views of the query. These approaches are not learning based, and require significant manual template-engineering efforts. Wong and Mooney [2007] present a statistical system that learns to generate sentences from lambda calculus expressions by inverting a semantic parser. Along similar lines, in Chapter 5 we present a neural system for generating textual descriptions for general SQL queries by learning from titles for SQL query posts from `stackoverflow.com`.

1.2 Challenges

Amongst the many challenges that models have to address for mapping language to general purpose source code, we explicitly address five main challenges in this dissertation: (1) Annotation Costs, (2) Context Dependence, (3) Scaling up Models, (4) Extensibility, and (5) Explainable NL to Code Systems.

1.2.1 Annotation Costs

Most statistical semantic parsers trained in the past have relied on the existence of datasets containing natural language carefully annotated with meaning representations by annotators with an adequate amount of linguistic training. While this results in effective parsers, the costs of creating these datasets are prohibitively high, which has always restricted their use exclusively to the research community and has hindered widespread adoption. There has been research on teaching models to induce these meaning representations as latent representations based on the output of execution of generated programs, but the very large space of possible programs typically either constrains these models to only produce short programs or severely increases decoding time. In this dissertation, we explore two relatively inexpensive ways of constructing datasets for NL to source code tasks: (1) In Chapter 2 we use NL-source code data from large scale online

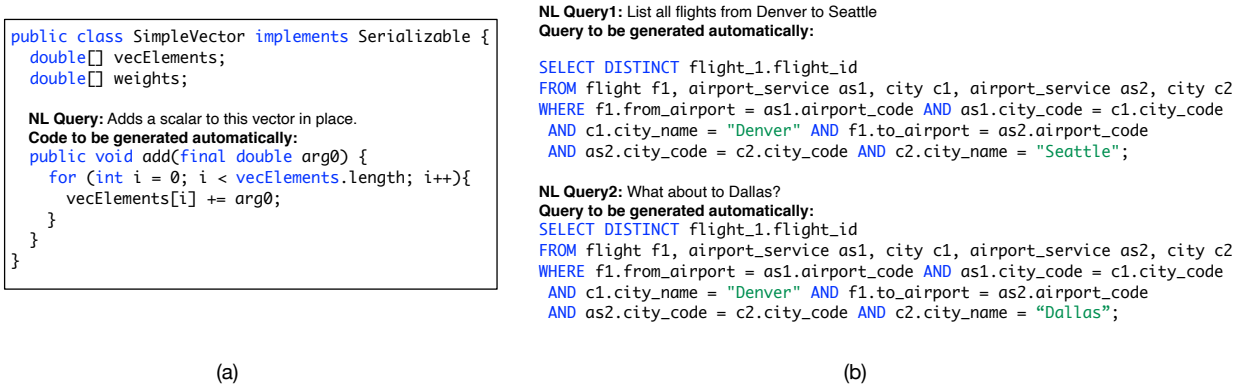

Figure 1.2: The target code in NL to source code tasks is heavily influenced by context. The structure of the `add` method in (a) depends on the type of the `vecElements` variable in the environment. Similarly, to generate the second SQL query in (b), models need to condition on both previous NL queries i.e. the interaction history thus far.

code corpora such as `github.com` as a means of model supervision, and, (2) In Chapter 3, we transform semantic parsers to use SQL (which is popularly used in the programming community) in place of specialized meaning representations, and we solicit annotations in an active manner from crowd programmers to minimize the number of examples annotated. We find that these strategies significantly reduce expenses associated with annotations and can improve widespread adoption of NL to code models.

1.2.2 Context Dependence

In this dissertation, we present models that map NL to source code for two main audiences viz. developers and non-expert users, and in both cases, the target source code is heavily dependent upon prior context or interaction history. For example, when producing code for developers, models need to take the existing programmatic context into account and the output code can significantly change when the context is different. In Figure 1.2 (a), the model needs to generate the `add` method from the NL query *Adds a scalar to this vector in place*, but this method directly depends on the datatype of the `vecElements` variable that is already defined in the class. Along similar lines, systems interacting with non-expert users need to take the interaction history with the user into account because it can resolve various NL ambiguities. In Figure 1.2 (b), the second SQL query can only be generated after considering the NL in both interactions. This heavy dependence on the context, where a small contextual change results in a large change in target code, is challenging for NL to code models. We address this challenge for models geared towards developers in

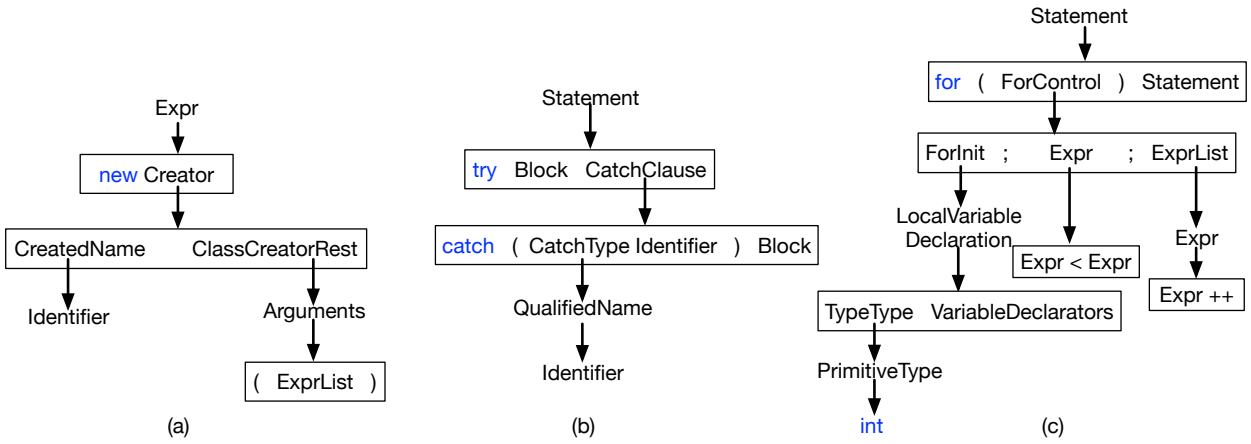

Figure 1.3: High-level code structures or programmatic idioms in Java, used frequently by programmers. (a) is an idiom to construct a new object with arguments, (b) represents a `try-catch` block, and, (c) is an integer-based `for` loop. In Chapter 4, we augment deep learning models with the ability to learn to apply such idioms for code generation.

Chapter 2 by encoding contextual elements and introducing attention mechanisms in the model decoder that attends to both the NL and the context together. We address the non-expert user case in Suhr et al. [2018] by conditioning on previous user interactions and reusing source code produced in previous interactions.

1.2.3 Scaling up Models

In Chapter 2 we find that for models that map NL to general purpose source code, syntax-based decoding is an invaluable method to ensure that the resulting source code is syntactically correct, and also contributes significantly to downstream performance. However, syntax-based decoding requires models to generate parse trees instead of source code tokens, and for real-world general purpose programming languages such as Java and Python, these parse trees can get significantly larger than the actual source code, making model training much slower (see Chapter 4). The increased length of decoded sequences, also directly affects accuracy, since it now also requires models to additionally encode very long-distance target side dependencies. In Chapter 4, we address this challenge by equipping models with the ability to generate code in terms of higher level source code structures i.e. programmatic idioms, paralleling the top down approach used by human programmers. We introduce general methods to extract idioms and use them directly in existing neural models without requiring any changes, and demonstrate their ability to help us significantly scale up NL to code models. Figure 1.3 presents examples of programmatic idioms that we automatically extract in Chapter

NL Query: Show me all papers from PLDI 2014
Query generated automatically:

```
SELECT DISTINCT paper.paperId FROM paper, venue
WHERE paper.venueId - venue.venueId AND venue.venueName = “pldi”;
```

Explanation:
Get me all papers from PLDI

Figure 1.4: Explaining the generated source code back to the user is vital in NL to code systems to establish user trust and to avoid misleading the user. The database querying system here ignores the year *2014* and returns results for all years. This anomaly can be hard for users to catch by just looking at the output.

4 from the CONCODE dataset, which are also common higher level elements that most programmers are familiar with.

1.2.4 Extensibility

Deployed NL to code systems have the additional challenge of dealing with user instructions that change and evolve over time to being distributionally different from data that the models were original trained on. Earlier statistical semantic parsers that mapped NL to specialized meaning representations were expensive to retrain and extend, and thus, lacked extensibility properties. In this dissertation, we advocate the use of general purpose popular programming languages such as SQL as an inexpensive alternative representation to train models for semantic parsing. Additionally, the ubiquitous nature of SQL also permits us to easily and swiftly solicit new annotations to periodically retrain deployed models to adapt to changing user input distributions.

1.2.5 Explainable NL to Code Systems

One main target audience for NL to code models is non-expert users, who are unfamiliar with writing programs and rely on these models to translate their intent to executable code. This also means that these users are not able to check the source code output by these models for correctness, and are therefore, typically only presented with the output of execution of the target source code. Since in a large number of cases, it is impossible to judge the correctness of the model by simply looking at the output, even after testing a handful of inputs, users of these models are typically left with a feeling of uncertainty and reservation. One solution

that can boost their confidence in the presented output is to train models that can explain generated source code back to the user. In Figure 1.4, the user wishes to see all *papers from PLDI 2014*, but the generated SQL query ignores the year *2014* and returns results for all years. This anomaly can be hard for users to catch by just looking at the output, and having an automatically generated explanation can help ascertain users of the results. In Chapter 5, we present neural models to explain the functionality of source code to users by learning to describe source code from code-related discussions on online code forums such as `stackoverflow.com`, as a step towards incorporating some level of explainability into deployed NL to code systems.

1.3 Dissertation Outline

This dissertation presents efficient deep learning models and training paradigms to map natural language in different contexts to general purpose source code, for applications intended for two main audiences viz. developers and non-expert users. Chapter 2 introduces a new task of contextual code generation from NL in programmatic environments. We present specialized neural attentional encoder-decoder architectures that exhibit contextual awareness while decoding syntactically correct source code, and leverage inexpensive annotations from existing large scale code corpora on the web to supervise these models. Chapter 3 presents inexpensive and efficient methods to learn natural language interfaces for arbitrary databases by training sequence to sequence models to directly map NL instructions to SQL queries. Using a combination of active learning and user feedback we reduce annotation costs and make models extensible over time. We next present the use of programmatic idioms to improve training speed and source code accuracy for both the prior tasks in Chapter 4. Finally, in order to build trust with their users, NL to code systems are required to explain the executed source code back to the users in order to avoid misleading them by presenting output different from the user's intent. To this end, Chapter 5 presents models that describe the functionality of source code in natural language. In the last chapter, we summarize follow-up research that has extended our work from this dissertation and also discuss potential areas for future work.

Chapter 2

Mapping Language to Code in Programmatic Context

Source code is rarely written in isolation. It depends significantly on the programmatic context, such as the class that the code would reside in. To study this phenomenon, in this Chapter, we introduce the task of generating class member functions given English documentation and the programmatic context provided by the rest of the class. This task is challenging because the desired code can vary greatly depending on the functionality the class provides (e.g., a sort function may or may not be available when we are asked to "return the smallest element" in a particular member variable list). We introduce CONCODE, a new large dataset with over 100,000 examples consisting of Java classes from online code repositories, and develop a new encoder-decoder architecture that models the interaction between the method documentation and the class environment. We also present a detailed error analysis suggesting that there is significant room for future work on this task.

2.1 Introduction

Natural language can be used to define complex computations that reuse the functionality of rich, existing code bases. However, existing approaches for automatically mapping natural language (NL) to executable code have considered limited language or code environments. They either assume fixed code templates (i.e.,

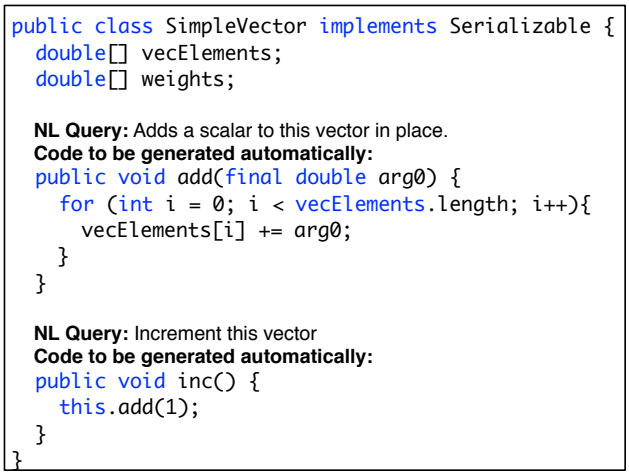

Figure 2.1: Code generation based on the class environment and method documentation. The figure shows a class where the programmer wants to automatically generate the `add` method from documentation, assuming the rest of the class is already written. The system needs to understand that `vecElements` is the vector to be augmented, and that the method must take in a scalar parameter as the element to be added. The model also needs to disambiguate between the member variables `vecElements` and `weights`.

generate only parts of a method with a predefined structure; Quirk et al., 2015), a fixed context (i.e., generate the body of the same method within a single fixed class; Ling et al., 2016), or no context at all (i.e., generate code tokens from the text alone; Oda et al., 2015). In this Chapter, we introduce new data and methods for learning to map language to source code within the context of a real-world programming environment, with application to generating member functions from documentation for automatically collected Java class environments.

The presence of rich context provided by an existing code environment better approximates the way programmers capitalize on code re-use, and also introduces new language understanding challenges. Models must (a) map the NL to environment variables, library API calls and user-defined methods found elsewhere in the class based on their names, types and signatures, and (b) decide on the structure of the resulting code. For example, in Figure 2.1, to generate the method `inc()` from the corresponding NL, *Increment this vector*, it is crucial to know of the existence of class method `add()`. This helps us decide if it should directly call `add()` or generate the method from scratch by iterating through the `vecElements` array and incrementing each element. Similarly, for generating the `add()` method, the code needs to use the class variable `vecElements` correctly. Overall, the code environment provides rich information relating to the intent of the developer, and can be used to considerably reduce ambiguity in the NL documentation.

To learn such a code generator, we use a specialized neural encoder-decoder model that (a) encodes the NL together with representations based on sub-word units for environment identifiers (member variables, methods) and data types, and (b) decodes the resulting code using an attention mechanism with multiple steps, by first attending to the NL, and then to the variables and methods, thus also learning to copy variables and methods. This two-step attention helps the model to match words in the NL with representations of the identifiers in the environment. Rather than directly generating the output source code tokens [Dong and Lapata, 2016; Iyer et al., 2017], the decoder generates production rules from the grammar of the target programming language similar to Rabinovich et al. [2017], Yin and Neubig [2017], and Krishnamurthy et al. [2017] and therefore, guarantees the syntactic well-formedness of the output.

To train our model, we collect and release CONCODE, a new dataset comprising over 100,000 (class environment, NL, code) tuples by gathering Java files containing method documentation from public Github repositories. This is an order of magnitude larger than existing datasets that map NL to source code for a general purpose language (MTG from Ling et al. [2016] has 13k examples), contains a larger variety of output code templates than existing datasets built for a specific domain, and is the first to condition on the environment of the output code. Also, by design, it contains examples from several domains, thus introducing open-domain challenges of new identifiers during test time (some e.g. class environments are LookupCommand, ColumnFileReader and ImageSequenceWriter). Our model achieves an exact match accuracy of 8.6% and a BLEU score (a metric for partial credit; Papineni et al., 2002) of 22.11, outperforming retrieval and recent neural methods. We also provide an extensive ablative analysis, quantifying the contributions that come from the context and the model, and suggesting that our work opens up various areas for future investigation. Later on, in Chapter 4 of this dissertation, we will further improve this model by making various simplifications in the encoder as well as by augmenting the decoder with programmatic idioms.

2.2 Task Definition

We introduce the task of generating source code from NL documentation, conditioned on the class environment the code resides in. The environment comprises two lists of entities: (1) class member variable names with their data types (for example, `double[] vecElements` as seen in Figure 2.2), and (2) member

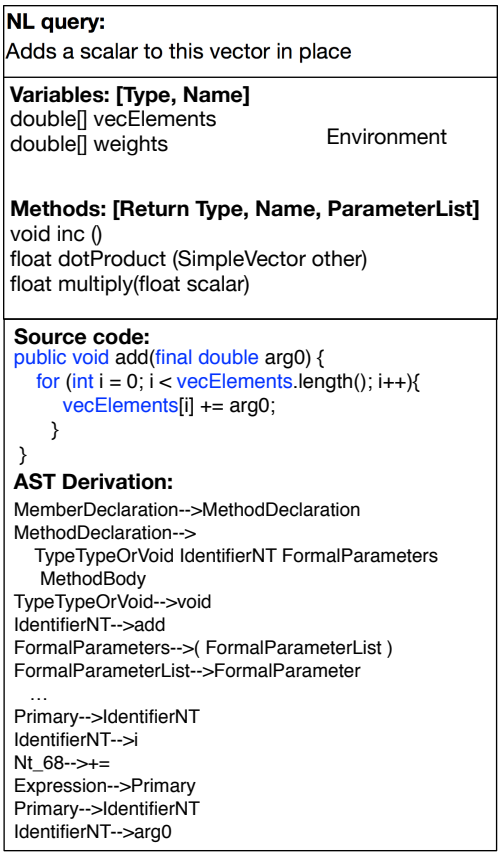

Figure 2.2: Our task involves generating the derivation of the source code of a method based on the NL documentation, class member variables (names and data types), and other class member methods (method names and return types), which form the code environment.

function names together with their return types (for example, `void inc()`).[1] Formally, let $q^{(i)}$, $a^{(i)}$ denote the NL and source code respectively for the i^{th} training example, where $a^{(i)}$ is a sequence of production rules that forms the derivation of its underlying source code. The environment comprises a list of variables names ${v^{(i)}}_{1..|v^{(i)}|}$ and their corresponding types ${t^{(i)}}_{1..|t^{(i)}|}$, as well as method names ${m^{(i)}}_{1..|m^{(i)}|}$ and their return types ${r^{(i)}}_{1..|r^{(i)}|}$. Our goal is to generate the derivation of $a^{(i)}$ given $q^{(i)}$ and the environment (see Figure 2.2).

[1]The method parameters and body can be used as well but we leave this to future work.

2.3 Models

We evaluate a number of encoder-decoder models that generate source code derivations from NL and the class environment. Our best model encodes all environment components broken down into sub-word units [Sennrich et al., 2016] separately, using Bi-LSTMs and decodes these contextual representations to produce a sequence of valid production rules that derive syntactically valid source code. The decoder also uses a two-step attention mechanism to match words in the NL with environment components, and then uses a supervised copy mechanism [Gu et al., 2016a] to incorporate environment elements in the resulting code. We describe this architecture below. In Chapter 4, we further improve this model by simplifying various aspects of the encoder, and also augment the decoder with programmatic idioms to push performance even higher.

2.3.1 Encoder

The encoder computes contextual representations of the NL and each component in the environment. Each word of the NL, q_i, is embedded into a high dimensional space using Identifier matrix I (denoted as $\mathbf{q_i}$) followed by the application of a n-layer bidirectional LSTM [Hochreiter and Schmidhuber, 1997]. The hidden states of the last layer $(h_1, \cdots, h_z)$ are passed on to the attention layer, while the hidden states at the last token are used to initialize the decoder.

$$h_1, \cdots, h_z = \text{BiLSTM}(\mathbf{q_1}, \ldots, \mathbf{q_z})$$

To encode variables and methods, the variable types (t_i) and method return types (r_i) are embedded using a type matrix T (denoted as $\mathbf{t_i}$ and $\mathbf{r_i}$). To encode the variable and method names (v_i, m_i), they are first split based on camel-casing, and each component is embedded using I, represented as $\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}}$ and $\mathbf{m_{i1}}, \ldots, \mathbf{m_{ik}}$. The encoded representation of the variable and method names is the final hidden state of the last layer of a Bi-LSTM over these embeddings ($\mathbf{v_i}$ and $\mathbf{m_i}$). Finally, a 2-step Bi-LSTM is executed on the concatenation of the variable type embedding and the variable name encoding. The corresponding hidden states form the final representations of the variable type and the variable name ($\hat{t_i}$, $\hat{v_i}$) and are passed on to the attention mechanism. Method return types and names are processed identically using the same

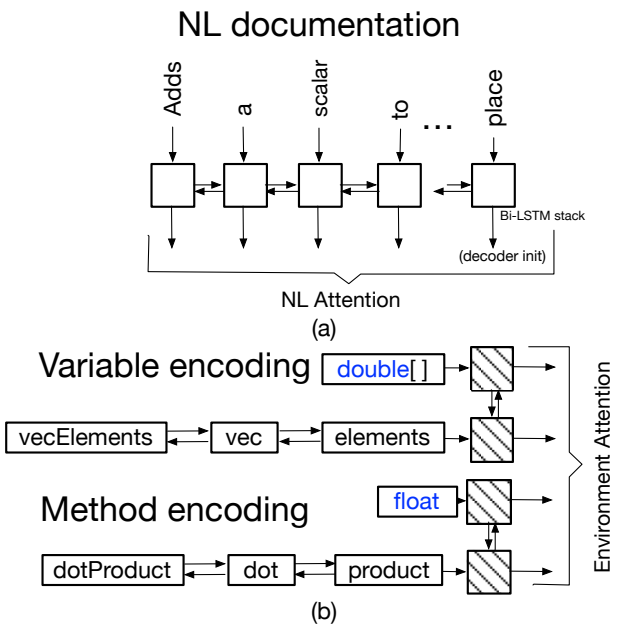

Figure 2.3: The encoder creates contextual representations of the NL (a), the variables and the methods (b). Variable (method) names are split based on camel-casing and encoded using a BiLSTM. The variable (method) type and name are further contextualized using another BiLSTM.

Bi-LSTMs and embedding matrices ($\hat{r_i}$, $\hat{m_i}$).

$$\mathbf{t_i} = t_i T;\ \mathbf{v_{ij}} = v_{ij} I$$

$$\mathbf{r_i} = r_i T;\ \mathbf{m_{ik}} = m_{ik} I$$

$$\mathbf{v_i} = \text{BiLSTM}(\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}})$$

$$\mathbf{m_i} = \text{BiLSTM}(\mathbf{m_{i1}}, \ldots, \mathbf{m_{ik}})$$

$$\hat{t_i}, \hat{v_i} = \text{BiLSTM}(\mathbf{t_i}, \mathbf{v_i})$$

$$\hat{r_i}, \hat{m_i} = \text{BiLSTM}(\mathbf{r_i}, \mathbf{m_i})$$

Figure 2.3 shows an example of the encoder.

2.3.2 Decoder

We represent the source code to be produced as a sequence of production rules (a_t at step t), with a non-terminal n_t on the left hand side and a combination of terminal and non-terminal symbols on the right

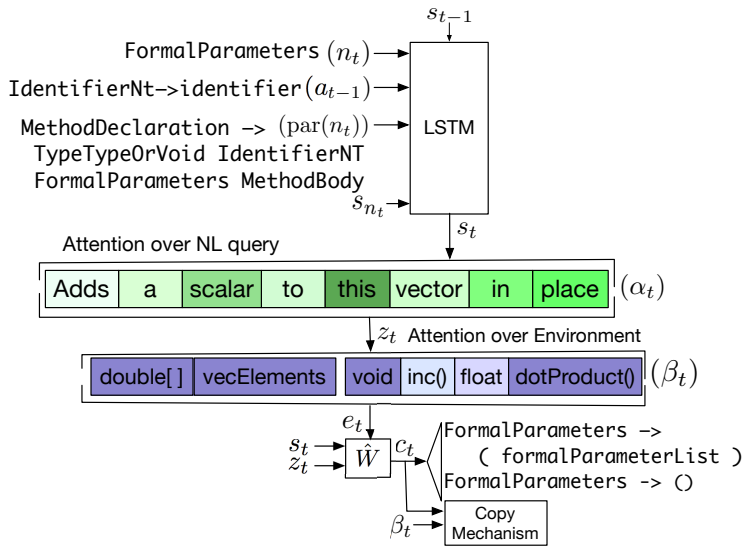

Figure 2.4: The hidden state s_t of our decoder is a function of the previous hidden state, current non-terminal, previous production rule, parent rule, and the parent hidden state. s_t is used to attend on the NL and compress it into z_t, which is then used to attend over the environment variables and methods to generate e_t. The decoder uses all these context vectors to produce a distribution over valid right hand side values of the current non-terminal, and also learns to copy from the environment.

hand side (see Figure 2.2). The first non-terminal is *MemberDeclaration*. Subsequently, every non-terminal is expanded in a depth first left to right fashion, similar to Yin and Neubig [2017]. The probability of a source code snippet is decomposed as a product of the conditional probability of generating each step in the sequence of rules conditioned on the previously generated rules. Our decoder is an LSTM-based RNN that produces a context vector c_t at each time step, which is used to compute a distribution over next actions.

$$p(a_t|a_{<t}) \propto exp(W^{n_t} c_t) \tag{2.1}$$

Here, W^{n_t} is a $|n_t| \times H$ matrix, where $|n_t|$ is the total number of unique production rules that n_t can be expanded to. The context vector c_t is computed using the hidden state s_t of an n-layer decoder LSTM cell and attention vectors over the NL and the context (z_t and e_t), as described below.

Decoder LSTM The decoder uses an n-layer LSTM whose hidden state s_t is computed based on the current non-terminal n_t to be expanded, the previous production rule a_{t-1}, the parent production rule $\text{par}(n_t)$ that produced n_t, the previous decoder LSTM state s_{t-1}, and the decoder state of the LSTM cell that

produced n_t, denoted as s_{n_t}.

$$s_t = \text{LSTM}(n_t, a_{t-1}, \text{par}(n_t), s_{t-1}, s_{n_t}) \tag{2.2}$$

We use an embedding matrix N to embed n_t and matrix A to embed a_{t-1} and par(n_t). If a_{t-1} is a rule that generates a terminal node that represents an identifier or literal, it is represented using a special rule *IdentifierOrLiteral* to collapse all these rules into a single previous rule.

Two-step Attention At time step t, the decoder first attends to every token in the NL representation, h_i, using the current decoder state, s_t, to compute a set of attention weights α_t, which are used to combine h_i into an NL context vector z_t. We use a general attention mechanism [Luong et al., 2015], extended to perform multiple steps.

$$\alpha_{t,i} = \frac{\exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}{\sum_i \exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}$$
$$z_t = \sum_i \alpha_{t,i} h_i$$

The context vector z_t is used to attend over every type (return type) and variable (method) name in the environment, to produce attention weights β_t that are used to combine the entire context $x = [t : v : r : m]$ into an environment context vector e_t.[2]

$$\beta_{t,j} = \frac{\exp(z_t^{\mathrm{T}} \mathbf{G} x_j)}{\sum_j \exp(z_t^{\mathrm{T}} \mathbf{G} x_j)}$$
$$e_t = \sum_j \beta_{t,j} x_j$$

Finally, c_t is computed using the decoder state and both context vectors z_t and e_t:

$$c_t = tanh(\hat{W}[s_t : z_t : e_t])$$

[2]":" denotes concatenation.

Supervised Copy Mechanism Since the class environment at test time can belong to previously unseen new domains, our model needs to learn to copy variables and methods into the output. We use the copying technique of Gu et al. [2016a] to compute a copy probability at every time step t using vector b of dimensionality H.

$$\text{copy}(t) = \sigma(b^T c_t)$$

Since we only require named identifiers or user defined types to be copied, both of which are produced by production rules with n_t as *IdentifierNT*, we make use of this copy mechanism only in this case. Identifiers can be generated by directly generating derivation rules (see equation 2.1), or by copying from the environment. The probability of copying an environment token x_j, is set to be the attention weights $\beta_{t,j}$ computed earlier, which attend exactly on the environment types and names which we wish to be able to copy. The copying process is supervised by preprocessing the production rules to recognize identifiers that can be copied from the environment, and both the generation and the copy probabilities are weighted by $1 - \text{copy}(t)$ and $\text{copy}(t)$ respectively. The LSTM decoder with attention mechanism is illustrated in Figure 2.4.

2.3.3 Baseline Models

Retrieval We evaluate a retrieval baseline, where the output source code for a test example is the training example source code whose NL is closest in terms of cosine similarity to the test NL using a tf-idf representation. We then replace all occurrences of environment variables and methods in the chosen training source code with similarly typed variables and methods from the environment of the test example, and we break ties randomly.

Seq2seq We evaluate a Seq2Seq baseline by representing the NL and context as a sequence formed by the concatenation of the NL, the variables and the methods with separators between them. The variables (methods) are represented with the type (return type) followed by the name, with a different separator between them. The encoder is an n-layer LSTM which initializes an LSTM-based decoder using its final hidden states. The decoder uses an attention mechanism [Luong et al., 2015] over the encoder states to

produce a conditional distribution over the next source code token (not production rule) given all the previous tokens. We replace UNK tokens in the output with source tokens having the most attention weight. We also attempted to evaluate the Seq2Tree model of Dong and Lapata [2016] but the redundancy in the model resulted in extremely long output sequences which did not scale. Experiments on a smaller dataset gave comparable results to Seq2seq.

Seq2prod This baseline corresponds to the action sequence model by Yin and Neubig [2017], with a BiLSTM over a sequence representation of the NL and context (same as Seq2seq), and a decoder that learns to generate a derivation of the AST of the underlying source code, similar to our model. The decoder uses the same attention mechanism as the Seq2seq, however, it uses supervised copying from the entire input sequence to handle unknown words encountered during testing.

2.4 CONCODE

We built a new dataset (CONCODE) from public Java projects on Github that contains environment information together with NL (Javadoc-style method comments) and code tuples. We gather Java files from $\sim$33,000 repositories, which are then split into train, development, and test sets based on repository, rather than purely randomly. Dividing based on the repository keeps the domains in the test set separate from the training set, therefore providing near zero-shot conditions that should truly test the ability of models to generalize to associate unseen NL tokens with previously unseen environment variables and methods. We also remove all examples from the development and test sets where the NL is exactly present in the training set. We further eliminate all Java classes that inherit from parent classes, since the resulting code may use variables and methods inherited from parent classes that reside in separate source files. While this is an important and interesting feature, we leave it for future work.

Every method that contains a Javadoc comment is treated as a training example. The Javadoc comment forms the NL, the method body is the target code to be generated, and all member variables, as well as other member method signatures are extracted and stored as part of the context. The Javadoc is preprocessed by stripping special symbols such as `@link`, `@code`, `@inheritDoc` and special fields such as `@params`. Methods that do not parse are eliminated and the rest are pre-processed by renaming locally defined variables

	Count
Train	100,000
Valid/Test	2,000/2,000
Average NL Length	13.73
Average Code Characters	119
Average Code Tokens	26.3
Average # Environment Variables	4.89
Average # Environment Methods	10.95
Average AST Nodes	79.6
# Node Types	153
# Production Rules	342*
% Getters	16.74%
% Setters	3.39%
% using Class Variables	68%
% using Class Methods	16.2%
% Unknown Types	7.65%

Table 2.1: Statistics of our dataset of (NL, context and code) tuples collected from Github. *Does not include rules that generate identifiers.

canonically, beginning at *loc0* and similarly for arguments, starting with *arg0*. We also replace all method names with the word *function* since it doesn't affect the semantics of the resulting program. Generating informative method names has been studied by Allamanis et al. [2015a]. We replace all string literals in the code with constants as they are often debug messages. Finally, we use an ANTLR java grammar[3] that is post-processed by adding additional non-terminals and rules to eliminate wildcard symbols in the grammar, in order to convert the source code into a sequence of production rules.

Using this procedure, we gather as many as 2 million (class environment, NL, code) training tuples and over 10,000 dev and test tuples. In order to speed up our development cycle and ensure that our models train in a reasonable amount of time, for this Chapter, we create our dataset (CONCODE) with 100,000 examples for training, and 2000 examples each for development and testing, respectively. Later in Chapter 4, we will use programmatic idioms to scale up our models to upto 500,000 training examples from our larger set. Table 2.1 summarizes the various statistics. We observe that on average, an environment contains $\sim$5 variables and $\sim$11 methods. Around 68% of the target code uses class member variables, and 16% of them use member methods, from the environment. Based on a frequency cutoff of 2 on the training set,

[3] `https://github.com/antlr/grammars-v4`

we find that 7% of the types in the development set code are unknown, hence they need to be copied from the environment. Since CONCODE is extracted from a diverse set of online repositories, there is a high variety of code templates in the dataset compared to existing datasets. For example, a random baseline on the Hearthstone card game dataset [Ling et al., 2016] gives a BLEU score of 40.3, but only a score of 10.2 on CONCODE. We release all code and data from this work.[4]

2.5 Experimental Setup

We restrict all models to examples where the length of the combination of the NL and the context is at most 200 tokens and the length of the output source code is at most 150 tokens. Source NL tokens are lower-cased, camel-cased identifiers are split and lower-cased, and are used together with the original version. The vocabulary for identifiers uses a frequency threshold of 7, resulting in a vocabulary of $32,600$ tokens. The types vocabulary uses a threshold of 2 resulting in $22,324$ types. We include all 153 non-terminals and 342 previous rules. We use a threshold of 2 for output production rules to filter out the long tail of rules creating identifiers and literals, resulting in $18,135$ output rules. Remaining values are replaced with the UNK symbol.

Hyperparameters We use an embedding size H of 1024 for identifiers and types. All LSTM cells use 2-layers and a hidden dimensionality of 1024 (512 on each direction for BiLSTMs). We use an embedding size of 512 for encoding non-terminals and rules in the decoder. We use dropout with $p = 0.5$ in between LSTM layers and at the output of the decoder over c_t. We train our model for 30 epochs using mini-batch gradient descent with a batch size of 20, and we use Adam [Kingma and Ba, 2015] with an initial learning rate of 0.001 for optimization. We decay our learning rate by 80% based on performance on the development set after every epoch.

Inference and Metrics Inference is done by first encoding the NL and context of the test example. We maintain a stack of symbols starting with the non-terminal, *MemberDeclaration*, and at each step, a non-terminal (terminals are added to the output) is popped off the stack to run a decoding step to generate the next

[4] `https://github.com/sriniiyer/concode`

Model	Exact	BLEU
Retrieval	2.25 (1.65)	20.27 (18.15)
Seq2Seq	3.2 (2.9)	23.51 (21.0)
Seq2Prod	6.65 (5.55)	21.29 (20.55)
Ours	8.6 (7.05)	22.11 (21.28)

Table 2.2: Exact match accuracy and BLEU score (for partial credit) on the test (development) set, comprising 2000 examples from previously unseen repositories.

Model	Exact	BLEU
Ours	7.05	21.28
-Variables	1.6	20.78
-Methods	6.2	21.74
-Two step attention	5.75	17.2
-Camel-case encoding	5.7	21.83

Table 2.3: Ablation of model features on the development set.

set of symbols to push onto the stack. The set of terminals generated along the way forms the output source code. We use beam search and maintain a ranked list of partial derivations (or code tokens for Seq2seq) at every step to explore alternate high-probability derivations. We use a beam size of 3 for all neural models. We copy over source tokens whenever preferred by the model output. We restrict the output to 150 tokens or 500 production rules.

To evaluate the quality of the output, we use Exact match accuracy between the reference and generated code. As a measure of partial credit, we also compute the BLEU score [Papineni et al., 2002], following recent work on code generation [Ling et al., 2016; Yin and Neubig, 2017]. BLEU is an n-gram precision-based metric that will be higher when more subparts of the predicted code match the provided reference.

2.6 Results

We present results for the context based code generation task on the test and dev sets in Table 2.2. Our model outperforms all baselines and achieves a gain of 1.95% exact match accuracy and 0.82 BLEU points, with respect to the next best models. The combination of independently encoding sub-word units and applying a two-step attention mechanism helps the model learn to better associate the correct variables/methods from the context and the language in the NL. Figure 2.5 (a) shows an example output of our model, which

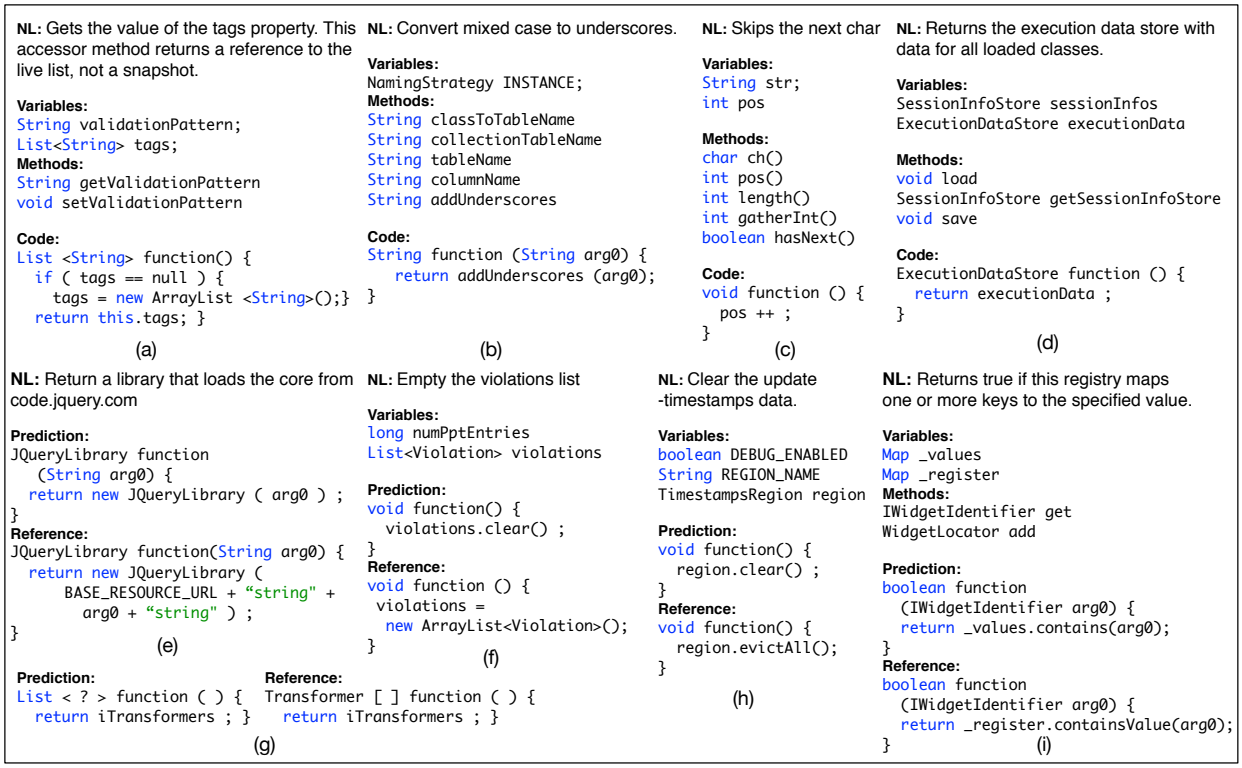

Figure 2.5: Analysis of our model output on development set examples. Some environment variables and methods are omitted for space. (a)-(d) represent cases where the model exactly produced the reference output. (e)-(g) are cases where the model output is very reasonable for a practical setting. In (f), the model produces a better solution than the reference. In (h), the context lacks information to produce the reference code. The model chooses the wrong element in (i) and could be improved by better encoder representations.

produces code structure intermixed with member variables (tags). In (b) our model learns to call method `addUnderscores` (an UNK in the vocabulary) with its correct return type (`String`). Similarly, in (d) our model also successfully learns to use a previously unseen type (`ExecutionDataStore`) when making use of the corresponding variable. (c) is an example of where the NL does not directly refer to the variable to be used. The mismatch between dev and test results is because we ensure that the dev and test examples come from non-overlapping Github repositories, resulting in different distributions.

Using a constrained decoder that generates syntax tree rules instead of tokens leads to significant gains in terms of exact match score (6.65 for Seq2prod vs 3.2 for Seq2seq), and shows that this is an important component of code generation systems. Seq2prod, however, fails on examples (b)-(d), since it is harder to learn to match the NL tokens with environment elements. Finally, all neural models outperform the retrieval baseline with member substitution.

To understand which components of the data and the model contribute most to the output, we perform an ablation study on the development set (Table 2.3). Removing the variables leads to a significant hit in exact match accuracy since 68% of examples (Table 2.1) refers to class variables; a similar but smaller reduction is incurred by removing methods. The presence of these components makes this task more challenging and also more aligned with programming scenarios in practice. Removing the two-step attention mechanism leads to a 1.3% drop in accuracy since the attention on the NL is unable to interact with the attention on the environment variables/methods. Removing camel-case encoding leads to a small drop mainly because many variable (method) names are single words.

2.7 Error Analysis

Subfigures 2.5(e)-(i) show cases where our model output did not exactly match the reference. In (e)-(g), the model output is semantically equivalent to the reference and is a very reasonable prediction in a practical setting. For example, in (e) the only difference between the prediction and the reference is the string concatenations to the url. Interestingly, in example (f) the prediction is a cleaner way to achieve the same effect as the reference, and this is a great example of the application of these models for suggesting standard and efficient code. Unfortunately, our model is penalized by the exact match metric here. Similarly, in (g), the model uses a generic list (`List<?>`) in place of the specific type (`Transformer[]`). Example (h) demonstrates a case where the model is unaware of methods that can be called on class members (specifically that `evictAll` is a member of the `TimestampsRegion` class), and requires augmenting the environment with additional member type documentation, which we believe will be an important area for future work. Example (i) calls for richer encoder representations, since our model incorrectly uses the `_values` variable instead of `_register`, as it is unable to associate the word "registry" with the right elements.

We further perform a qualitative analysis of 100 predictions on our development set (Table 2.4) and find that there is significant room for improvement with 71% of the predictions differing significantly from their references. 16% of the predictions are very close to their references with the difference being 1-2 tokens, while 11% are exactly correct. 2% of the predictions were semantically equivalent but not exactly equal to their references. In Chapter 4, we will considerably improve this model by simplifying the encoder and

Category	Fraction
Totally Wrong	62%
Marginally Correct	9%
Mostly Correct	16%
Exact Match	11%
Semantically Equivalent	2%

Table 2.4: Qualitative distribution of errors on the development set.

augmenting the decoder with programmatic idioms.

2.8 Related Work

There is significant existing research on mapping NL directly to executable programs in the form of logical forms [Zettlemoyer and Collins, 2005], λ-DCS [Liang et al., 2013], regular expressions [Kushman and Barzilay, 2013; Locascio et al., 2016], database queries [Iyer et al., 2017; Zhong et al., 2017] and general purpose programs [Balog et al., 2016; Allamanis et al., 2015b]. Ling et al. [2016] generate Java and Python source code from NL for card games, conditioned on categorical card attributes. Manshadi et al. [2013] generate code based on input/output examples for applications such as database querying. Gu et al. [2016b] use neural models to map NL queries to a sequence of API calls, and Neelakantan et al. [2015] augment neural models with a small set of basic arithmetic and logic operations to generate more meaningful programs. In this Chapter, we introduce a new task of generating programs from NL based on the environment in which the generated code resides, following the frequently occurring pattern in large repositories where the code depends on the types and availability of variables and methods in the environment.

Neural encoder-decoder models have proved effective in mapping NL to logical forms and also for directly producing general purpose programs. Ling et al. [2016] use a sequence-to-sequence model with attention and a copy mechanism to generate source code. Instead of directly generating a sequence of code tokens, recent methods focus on constrained decoding mechanisms to generate syntactically correct output using a decoder that is either grammar-aware or has a dynamically-determined modular structure paralleling the structure of the abstract syntax tree (AST) of the code [Dong and Lapata, 2016; Rabinovich et al., 2017; Krishnamurthy et al., 2017; Yin and Neubig, 2017]. Our model also uses a grammar-aware decoder similar

to Yin and Neubig [2017] to generate syntactically valid parse trees, augmented with a two-step attention mechanism [Chen et al., 2016b], followed by a supervised copying mechanism [Gu et al., 2016a] over the class environment.

Recent models for mapping NL to code have been evaluated on datasets containing highly templated code for card games (Hearthstone & MTG; Ling et al., 2016), or manually labeled per-line comments (DJANGO; Oda et al., 2015). These datasets contain $\sim$20,000 programs with short textual descriptions possibly paired with categorical data, whose values need to be copied onto the resulting code from a single domain. In this Chapter, we collect a new dataset of over 100,000 NL and code pairs, together with the corresponding class environment. Each environment and NL describe a specific domain and the dataset comprises thousands of different domains, that poses additional challenges. Having an order of magnitude more data than existing datasets makes training deep neural models very effective, as we saw in the experimental evaluation. While massive amounts of Github code have been used before for creating datasets on source code only [Allamanis and Sutton, 2013, 2014; Allamanis et al., 2016], we instead extract from Github a dataset of NL and code with an emphasis on context, in order to learn to map NL to code within a class.

2.9 Conclusion

In this Chapter, we introduce new data and methods for learning to generate source code from language within the context of a real-world code base. To train models for this task, we release CONCODE, a large new dataset of NL, code, and context tuples gathered inexpensively from online repositories, featuring code from a variety of domains. We also introduced a new encoder decoder model with a specialized context encoder which outperforms strong neural baselines by 1.95% exact match accuracy. Finally, analysis suggests that even richer models of programmatic context could further improve these results. We will revisit this task and model in Chapter 4, where we will improve performance considerably by simplifying the encoder and augmenting the decoder with programmatic idioms.

Chapter 3

Learning a Semantic Parser from User Feedback

While the previous chapter focused on NL to code models for developers, we now switch gears to describe models that will cater to non-expert users. In this Chapter, we present an approach to rapidly and easily build natural language interfaces to databases for new domains, whose performance improves over time based on user feedback, and requires minimal intervention. To achieve this, we adapt neural sequence models to map utterances directly to SQL with its full expressivity, bypassing any intermediate meaning representations. These models are immediately deployed online to solicit feedback from real users to flag incorrect queries. Finally, the popularity of SQL facilitates gathering annotations for incorrect predictions using the crowd, which is directly used to improve our models. This complete feedback loop, without intermediate representations or database specific engineering, opens up new ways of building high quality semantic parsers. Experiments suggest that this approach can be deployed quickly for any new target domain, as we show by learning a semantic parser for an online academic database from scratch.

3.1 Introduction

Existing semantic parsing approaches for building natural language interfaces to databases (NLIDBs) either use special-purpose intermediate meaning representations that lack the full expressivity of database query

Most recent papers of Michael I. Jordan

```
SELECT paper.paperId, paper.year
FROM paper, writes, author
WHERE paper.paperId = writes.paperId
  AND writes.authorId = author.authorId
  AND author.authorName = "michael i. jordan"
  AND paper.year =
    (SELECT max(paper.year)
     FROM paper, writes, author
     WHERE paper.paperId = writes.paperId
      AND writes.authorId = author.authorId
      AND author.authorName = "michael i. jordan");
```

I'd like to book a flight from San Diego to Toronto

```
SELECT DISTINCT f1.flight_id
FROM flight f1, airport_service a1, city c1,
  airport_service a2, city c2
WHERE f1.from_airport = a1.airport_code
  AND a1.city_code = c1.city_code
  AND c1.city_name = 'san diego'
  AND f1.to_airport = a2.airport_code
  AND a2.city_code = c2.city_code
  AND c2.city_name = 'toronto';
```

Figure 3.1: Utterances with corresponding SQL queries to answer them for two domains, an academic database and a flight reservation database.

languages or require extensive feature engineering, making it difficult to deploy them in new domains. In this chapter, we present a robust approach to quickly and easily learn and deploy semantic parsers from scratch, whose performance improves over time based on user feedback, and requires very little expert intervention. To learn these semantic parsers, we (1) adapt neural sequence models to map utterances directly to SQL thereby bypassing intermediate representations and taking full advantage of SQL's querying capabilities, (2) immediately deploy the model online to solicit questions and user feedback on results to reduce SQL annotation efforts, and (3) use crowd workers from skilled markets to provide SQL annotations that can directly be used for model improvement, in addition to being easier and cheaper to obtain than logical meaning representations. We demonstrate the effectiveness of the complete approach by successfully learning a semantic parser for an academic domain by simply deploying it online for three days.

This type of interactive learning is related to a number of recent ideas in semantic parsing, including batch learning of models that directly produce programs (e.g., regular expressions [Locascio et al., 2016]), learning from paraphrases (often gathered through crowdsourcing [Wang et al., 2015]), data augmentation (e.g. based on manually engineered semantic grammars [Jia and Liang, 2016]) and learning through direct

interaction with users (e.g., where a single user teaches the model new concepts [Wang et al., 2016]). However, there are unique advantages to our approach, including showing (1) that non-linguists can write SQL to encode complex, compositional computations (see Fig 3.1 for an example), (2) that external paraphrase resources and the structure of facts from the target database itself can be used for effective data augmentation, and (3) that actual database users can effectively drive the overall learning by simply providing feedback about what the model is currently getting correct.

Our experiments measure the performance of these learning advances, both in batch on existing datasets and through a simple online experiment for the full interactive setting. For the batch evaluation, we use sentences from the benchmark GeoQuery and ATIS domains, converted to contain SQL meaning representations. Our neural learning with data augmentation achieves reasonably high accuracies, despite the extra complexities of mapping directly to SQL. We also perform simulated interactive learning on this data, showing that with perfect user feedback our full approach could learn high quality parsers with only 55% of the data. Finally, we do a small scale online experiment for a new domain, academic paper metadata search, demonstrating that actual users can provide useful feedback and our full approach is an effective method for learning a high quality parser that continues to improve over time as it is used.

3.2 Related Work

Although diverse meaning representation languages have been used with semantic parsers – such as regular expressions [Kushman and Barzilay, 2013; Locascio et al., 2016], Abstract Meaning Representations (AMR) [Artzi et al., 2015; Misra and Artzi, 2016], and systems of equations [Kushman et al., 2014; Roy et al., 2016] – parsers for querying databases have typically used either logic programs [Zelle and Mooney, 1996], lambda calculus [Zettlemoyer and Collins, 2005], or λ-DCS [Liang et al., 2013] as the meaning representation language. All three of these languages are modeled after natural language to simplify parsing. However, none of them are used to query databases outside of the semantic parsing literature; therefore, they are understood by few people and not supported by standard database implementations. In contrast, we parse directly to SQL, which is a popular database query language with wide usage and support. Learning parsers directly from SQL queries has the added benefit that we can potentially hire programmers on skilled-labor

crowd markets to provide labeled examples, such as UpWork[1], which we demonstrate in this work.

A few systems have been developed to directly generate SQL queries from natural language [Popescu et al., 2003; Giordani and Moschitti, 2012; Poon, 2013]. However, all of these systems make strong assumptions on the structure of queries: they use manually engineered rules that can only generate a subset of SQL, require lexical matches between question tokens and table/column names, or require questions to have a certain syntactic structure. In contrast, our approach can generate arbitrary SQL queries, only uses lexical matching for entity names, and does not depend on syntactic parsing.

We use a neural sequence-to-sequence model to directly generate SQL queries from natural language questions. This approach builds on recent work demonstrating that such models are effective for tasks such as machine translation [Bahdanau et al., 2015] and natural language generation [Kiddon et al., 2016]. Recently, neural models have been successfully applied to semantic parsing with simpler meaning representation languages [Dong and Lapata, 2016; Jia and Liang, 2016] and short regular expressions [Locascio et al., 2016]. Our work extends these results to the task of SQL generation. Finally, Ling et al. [2016] generate Java/Python code for trading cards given a natural language description; however, this system suffers from low overall accuracy.

A final direction of related work studies methods for reducing the annotation effort required to train a semantic parser. Semantic parsers have been trained from various kinds of annotations, including labeled queries [Zelle and Mooney, 1996; Wong and Mooney, 2007; Zettlemoyer and Collins, 2005], question/answer pairs [Liang et al., 2013; Kwiatkowski et al., 2013; Berant et al., 2013], distant supervision [Krishnamurthy and Mitchell, 2012; Choi et al., 2015], and binary correct/incorrect feedback signals [Clarke et al., 2010; Artzi and Zettlemoyer, 2013]. Each of these schemes presents a particular trade-off between annotation effort and parser accuracy; however, recent work has suggested that labeled queries are the most effective [Yih et al., 2016]. Our approach trains on fully labeled SQL queries to maximize accuracy, but uses binary feedback from users to reduce the number of queries that need to be labeled. Annotation effort can also be reduced by using crowd workers to paraphrase automatically generated questions [Wang et al., 2015]; however, this approach may not generate the questions that users actually want to ask the database – an experiment in this paper demonstrated that 48% of users' questions in a calendar domain could not be

[1] `http://www.upwork.com`

generated.

3.3 Feedback-based Learning

Our feedback-based learning approach can be used to quickly deploy semantic parsers to create NLIDBs for any new domain. It is a simple interactive learning algorithm that deploys a preliminary semantic parser, then iteratively improves this parser using user feedback and selective query annotation. A key requirement of this algorithm is the ability to cheaply and efficiently annotate queries for chosen user utterances. We address this requirement by developing a model that directly outputs SQL queries (Section 3.4), which can also be produced by crowd workers.

```
Procedure LEARN(schema)
    T ← initial_data(schema)
    while true do
        𝒯 ← T ∪ paraphrase(T)
        𝒩 ← train_model(𝒯)
        for n ∈ new utterances do
            q ← predict(𝒩, n)
            ℛ ← execute(q)
            f ← feedback(ℛ)
            if f = correct then
                T ← T ∪ (n, q)
            else if f = wrong then
                q̂ ← annotate(n)
                T ← T ∪ (n, q̂)
            end
        end
    end
end
```

Algorithm 1: Feedback-based learning.

Our algorithm alternates between stages of training the model and making predictions to gather user feedback, with the goal of improving performance in each successive stage. The procedure is described in Algorithm 1. Our neural model $\mathscr{N}$ is initially trained on synthetic data T generated by domain-independent schema templates (see Section 3.4), and is then ready to answer new user questions, n. The results $\mathscr{R}$ of executing the predicted SQL query q are presented to the user who provides a binary correct/incorrect feedback signal. If the user marks the result correct, the pair (n, q) is added to the training set. If the user

marks the result incorrect, the algorithm asks a crowd worker to annotate the utterance with the correct query, $\hat{q}$, and adds $(n, \hat{q})$ to the training set. This procedure can be repeated indefinitely, ideally increasing parser accuracy and requesting fewer annotations in each successive stage.

3.4 Semantic Parsing to SQL

We use a neural sequence-to-sequence model for mapping natural language questions directly to SQL queries and this allows us to scale our feedback-based learning approach, by easily crowdsourcing labels when necessary. We further present two data augmentation techniques which use content from the database schema and external paraphrase resources.

3.4.1 Model

We use an encoder-decoder model with global attention, similar to Luong et al. [2015], where the anonymized utterance (see Section 3.4.2) is encoded using a bidirectional LSTM network, then decoded to directly predict SQL query tokens. Fixed pre-trained word embeddings from word2vec [Mikolov et al., 2013b] are concatenated to the embeddings that are learned for source tokens from the training data. The decoder predicts a conditional probability distribution over possible values for the next SQL token given the previous tokens using a combination of the previous SQL token embedding, attention over the hidden states of the encoder network, and an attention signal from the previous time step.

Formally, if $\mathbf{q_i}$ represents an embedding for the ith SQL token q_i, the decoder distribution is

$$p(q_i|q_1, \ldots, q_{i-1}) \propto \exp\left(\mathbf{W} \tanh(\hat{\mathbf{W}}[\mathbf{h_i} : \mathbf{c_i}])\right)$$

where $\mathbf{h_i}$ represents the hidden state output of the decoder LSTM at the i^{th} timestep, $\mathbf{c_i}$ represents the context vector generated using an attention weighted sum of encoder hidden states based on $\mathbf{h_i}$, and, $\mathbf{W}$ and $\hat{\mathbf{W}}$ are linear transformations. If $\mathbf{s_j}$ is the hidden representation generated by the encoder for the jth word in

the utterance (k words long), then the context vectors are defined to be:

$$\mathbf{c_i} = \sum_{j=1}^{k} \alpha_{i,j} \cdot \mathbf{s_j}$$

The attention weights $\alpha_{i,j}$ are computed using an inner product between the decoder hidden state for the current timestep $\mathbf{h_i}$, and the hidden representation of the jth source token $\mathbf{s_j}$:

$$\alpha_{i,j} = \frac{\exp(\mathbf{h_i}^{\mathrm{T}} \mathbf{F} \mathbf{s_j})}{\sum_{j=1}^{k} \exp(\mathbf{h_i}^{\mathrm{T}} \mathbf{F} \mathbf{s_j})}$$

where $\mathbf{F}$ is a linear transformation. The decoder LSTM cell f computes the next hidden state $\mathbf{h_i}$, and cell state, $\mathbf{m_i}$, based on the previous hidden and cell states, $\mathbf{h_{i-1}}, \mathbf{m_{i-1}}$, the embeddings of the previous SQL token $\mathbf{q_{i-1}}$ and the context vector of the previous timestep, $\mathbf{c_{i-1}}$

$$\mathbf{h_i}, \mathbf{m_i} = f(\mathbf{h_{i-1}}, \mathbf{m_{i-1}}, \mathbf{q_{i-1}}, \mathbf{c_{i-1}})$$

We apply dropout on non-recurrent connections for regularization, as suggested by Pham et al. [2014]. Beam search is used for decoding the SQL queries after learning.

3.4.2 Entity Anonymization

We handle entities in the utterances and SQL by replacing them with their types, using incremental numbering to model multiple entities of the same type (e.g., `CITY_NAME_1`). During training, when the SQL is available, we infer the type from the associated column name; for example, Boston is a city in `city.city_name = 'Boston'`. To recognize entities in the utterances at test time, we build a search engine on all entities from the target database. For every span of words (starting with a high span size and progressively reducing it), we query the search engine using a TF-IDF scheme to retrieve the entity that most closely matches the span, then replace the span with the entity's type. We store these mappings and apply them to the generated SQL to fill in the entity names. TF-IDF matching allows some flexibility in matching entity names in utterances, for example, a user could say *Donald Knuth* instead of *Donald E. Knuth*.

```
Get all <ENT1>.<NAME> having
<ENT2>.<COL1>.<NAME> as <ENT2>.<COL1>.<TYPE>

SELECT <ENT1>.<DEF> FROM JOIN_FROM(<ENT1>, <ENT2>)
WHERE JOIN_WHERE(<ENT1>, <ENT2>) AND
 <ENT2>.<COL1> = <ENT2>.<COL1>.<TYPE>
```

(a) Schema template

```
Get all author having dataset  as DATASET_TYPE

SELECT author.authorId
FROM author , writes , paper , paperDataset , dataset
WHERE author.authorId = writes.authorId
 AND writes.paperId = paper.paperId
 AND paper.paperId = paperDataset.paperId
 AND paperDataset.datasetId = dataset.datasetId
 AND dataset.datasetName = DATASET_TYPE
```

(b) Generated utterance-SQL pair

Figure 3.2: (a) Example schema template consisting of a question and SQL query with slots to be filled with database entities, columns, and values; (b) Entity-anonymized training example generated by applying the template to an academic database.

3.4.3 Data Augmentation

We present two data augmentation strategies that either (1) provide the initial training data to start the interactive learning, before more labeled examples become available, or (2) use external paraphrase resources to improve generalization.

Schema Templates To bootstrap the model to answer simple questions initially, we defined 22 language/SQL templates that are schema-agnostic, so they can be applied to any database. These templates contain slots whose values are populated given a database schema. An example template is shown in Figure 3.2a. The `<ENT>` types represent tables in the database schema, `<ENT>.<COL>` represents a column in the particular table and `<ENT>.<COL>.<TYPE>` represents the type associated with the particular column. A template is instantiated by first choosing the entities and attributes. Next, join conditions, i.e., `JOIN_FROM` and `JOIN_WHERE` clauses, are generated from the tables on the shortest path between the chosen tables in the database schema graph, which connects tables (graph nodes) using foreign key constraints. Figure 3.2b shows an instantiation of a template using the path author - writes - paper - paperdataset - dataset. SQL queries generated in this manner are guaranteed to be executable on the target database. On the language side, an English name of each entity is plugged into the template to generate an utterance for the query.

Paraphrasing The second data augmentation strategy uses the Paraphrase Database (PPDB) [Ganitkevitch et al., 2013] to automatically generate paraphrases of training utterances. Such methods have been recently used to improve performance for parsing to logical forms [Chen et al., 2016a]. PPDB contains over 220 million paraphrase pairs divided into 6 sets (small to XXXL) based on precision of the paraphrases. We use the one-one and one-many paraphrases from the large version of PPDB. To paraphrase a training utterance, we pick a random word in the utterance that is not a stop word or entity and replace it with a random paraphrase. We perform paraphrase expansion on all examples labeled during learning, as well as the initial seed examples from schema templates.

3.5 Benchmark Experiments

Our first set of experiments demonstrates that our semantic parsing model has comparable accuracy to previous work, despite the increased difficulty of directly producing SQL. We demonstrate this result by running our model on two benchmark datasets for semantic parsing, GEO880 and ATIS.

3.5.1 Data sets

GEO880 is a collection of 880 utterances issued to a database of US geographical facts (Geobase), originally in Prolog format. Popescu et al. [2003] created a relational database schema for Geobase together with SQL queries for a subset of 700 utterances. To compare against prior work on the full corpus, we annotated the remaining utterances and used the standard 600/280 training/test split [Zettlemoyer and Collins, 2005].

ATIS is a collection of 5,418 utterances to a flight booking system, accompanied by a relational database and SQL queries to answer the questions. We use 4,473 utterances for training, 497 for development and 448 for test, following Kwiatkowski et al. [2011]. The original SQL queries were very inefficient to execute due to the use of `IN` clauses, so we converted them to joins [Ramakrishnan and Gehrke, 2003] while verifying that the output of the queries was unchanged.

Table 3.1 shows characteristics of both data sets. GEO880 has shorter queries but is more compositional: almost 40% of the SQL queries have at least one nested subquery. ATIS has the longest utterances and queries, with an average utterance length of 11 words and an average SQL query length of 67 tokens. They also operate on approximately 6 tables per query on average. We release our processed versions of both

	Geo880	ATIS	SCHOLAR
Avg. NL length	7.56	10.97	6.69
NL vocab size	151	808	303
Avg. SQL length	16.06	67.01	28.85
SQL vocab size	89	605	163
% Subqueries > 1	39.8	12.42	2.58
# Tables	1.19	5.88	3.33

Table 3.1: Utterance and SQL query statistics for each dataset. Vocabulary sizes are counted after entity anonymization.

datasets.

3.5.2 Experimental Methodology

We follow a standard train/dev/test methodology for our experiments. The training set is augmented using schema templates and 3 paraphrases per training example, as described in Section 3.4. Utterances were anonymized by replacing them with their corresponding types and all words that occur only once were replaced by UNK symbols. The development set is used for hyperparameter tuning and early stopping. For GEO880, we use cross validation on the training set to tune hyperparameters. We used a minibatch size of 100 and used Adam [Kingma and Ba, 2015] with a learning rate of 0.001 for 70 epochs for all our experiments. We used a beam size of 5 for decoding. We report test set accuracy of our SQL query predictions by executing them on the target database and comparing the result with the true result.

3.5.3 Results

Tables 3.2 and 3.3 show test accuracies based on denotations for our model on GEO880 and ATIS respectively, compared with previous work.[2] To our knowledge, this is the first result on directly parsing to SQL to achieve comparable performance to prior work without using any database-specific feature engineering. Popescu et al. [2003] and Giordani and Moschitti [2012] also directly produce SQL queries but on a subset of 700 examples from GEO880. The former only works on semantically tractable utterances where words can be unambiguously mapped to schema elements, while the latter uses a reranking approach that also limits the complexity of SQL queries that can be handled. GUSP [Poon, 2013] creates an intermediate rep-

[2]Note that 2.8% of GEO880 and 5% ATIS gold test set SQL queries (before any processing) produced empty results.

System	Acc.
Ours (SQL)	82.5
Popescu et al. [2003] (SQL)	77.5*
Giordani and Moschitti [2012] (SQL)	87.2*
Dong and Lapata [2016]	84.6◇†
Jia and Liang [2016]	89.3◇
Liang et al. [2013]	91.1◇

Table 3.2: Accuracy of SQL query results on the Geo880 corpus; * use Geo700; ◇ convert to logical forms instead of SQL; † measure accuracy in terms of obtaining the correct logical form, other systems, including ours, use denotations.

System	Acc.
Ours (SQL)	79.24
GUSP [Poon, 2013] (SQL)	74.8
GUSP++ [Poon, 2013] (SQL)	83.5
Zettlemoyer and Collins [2007]	84.6◇†
Dong and Lapata [2016]	84.2◇†
Jia and Liang [2016]	83.3◇
Wang et al. [2014]	91.3◇†

Table 3.3: Accuracy of SQL query results on ATIS; ◇ convert to logical forms instead of SQL; † measure accuracy in terms of obtaining the correct logical form, other systems, including ours, use denotations.

resentation that is then deterministically converted to SQL to obtain an accuracy of 74.8% on ATIS, which is boosted to 83.5% using manually introduced disambiguation rules. However, it requires a lot of SQL specific engineering (for example, special nodes for argmax) and is hard to extend to more complex SQL queries.

On both datasets, our SQL model achieves reasonably high accuracies approaching that of the best non-SQL results. Most relevant to this work are the neural sequence based approaches of Dong and Lapata [2016] and Jia and Liang [2016]. We note that Jia and Liang [2016] use a data recombination technique that boosts accuracy from 85.0 on GEO880 and 76.3 on ATIS; this technique is also compatible with our model and we hope to experiment with this in future work. Our results demonstrate that these models are powerful enough to directly produce SQL queries. Thus, our methods enable us to utilize the full expressivity of the SQL language without any extensions that certain logical representations require to answer more complex queries.

System	GEO880	ATIS
Ours	84.8	86.2
- paraphrases	81.8	84.3
- templates	84.7	85.7

Table 3.4: Addition of paraphrases to the training set helps performance, but template based data augmentation does not significantly help in the fully supervised setting. Accuracies are reported on the standard dev set for ATIS and on the training set, using cross-validation, for Geo880.

More importantly, it can be immediately deployed for users in new domains, with a large programming community available for annotation, and thus, fits effectively into a framework for interactive learning.

We perform ablation studies on the development sets (see Table 3.4) and find that paraphrasing using PPDB consistently helps boost performance. However, unlike in the interactive experiments (Section 3.6), data augmentation using schema templates does not improve performance in the fully supervised setting.

3.6 Interactive Learning Experiments

In this section, we learn a semantic parser for an academic domain from scratch by deploying an online system using our interactive learning algorithm (Section 3.3). After three train-deploy cycles, the system correctly answered 63.51% of user's questions. To our knowledge, this is the first effort to learn a semantic parser using a live system, and is enabled by our models that can directly parse language to SQL without manual intervention.

3.6.1 User Interface

We developed a web interface for accepting natural language questions to an academic database from users, using our model to generate a SQL query, and displaying the results after execution. Several example utterances are also displayed to help users understand the domain. Together with the results of the generated SQL query, users are prompted to provide feedback which is used for interactive learning. Figures 3.3 and 3.4 present screenshots of our user interface.

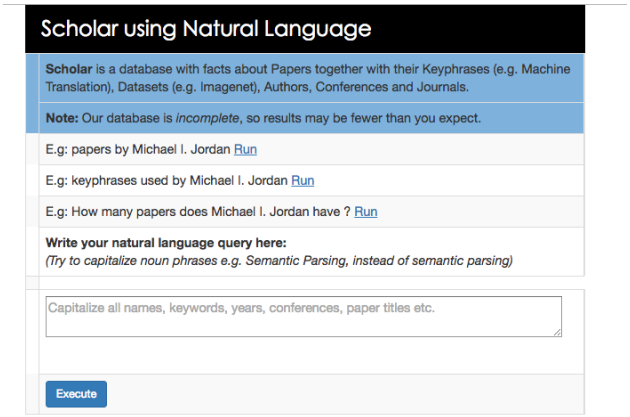

Figure 3.3: Users were presented with example utterances and a text box to enter their own utterance.

Collecting accurate user feedback on predicted queries is a key challenge in the interactive learning setting for two reasons. First, the system's results can be incorrect due to poor entity identification or incompleteness in the database, neither of which are under the semantic parser's control. Second, it can be difficult for users to determine if the presented results are in fact correct. This determination is especially challenging if the system responds with the correct type of result, for example, if the user requests "papers at ACL 2016" and the system responds with all ACL papers.

We address this challenge by providing users with two assists for understanding the system's behavior, and allowing users to provide more granular feedback than simply correct/incorrect. The first assist is **type highlighting**, which highlights entities identified in the utterance, for example, "paper by *Michael I. Jordan (AUTHOR)* in *ICRA (VENUE)* in *2016 (YEAR)*." This assist is especially helpful because the academic database contains noisy keyword and dataset tables that were automatically extracted from the papers. The second assist is **utterance paraphrasing**, which shows the user another utterance that maps to the same SQL query. For example, for the above query, the system may show "what papers does *Michael I. Jordan (AUTHOR)* have in *ICRA (VENUE)* in *2016 (YEAR)*." This assist only appears if a matching query (after entity anonymization) exists in the model's training set.

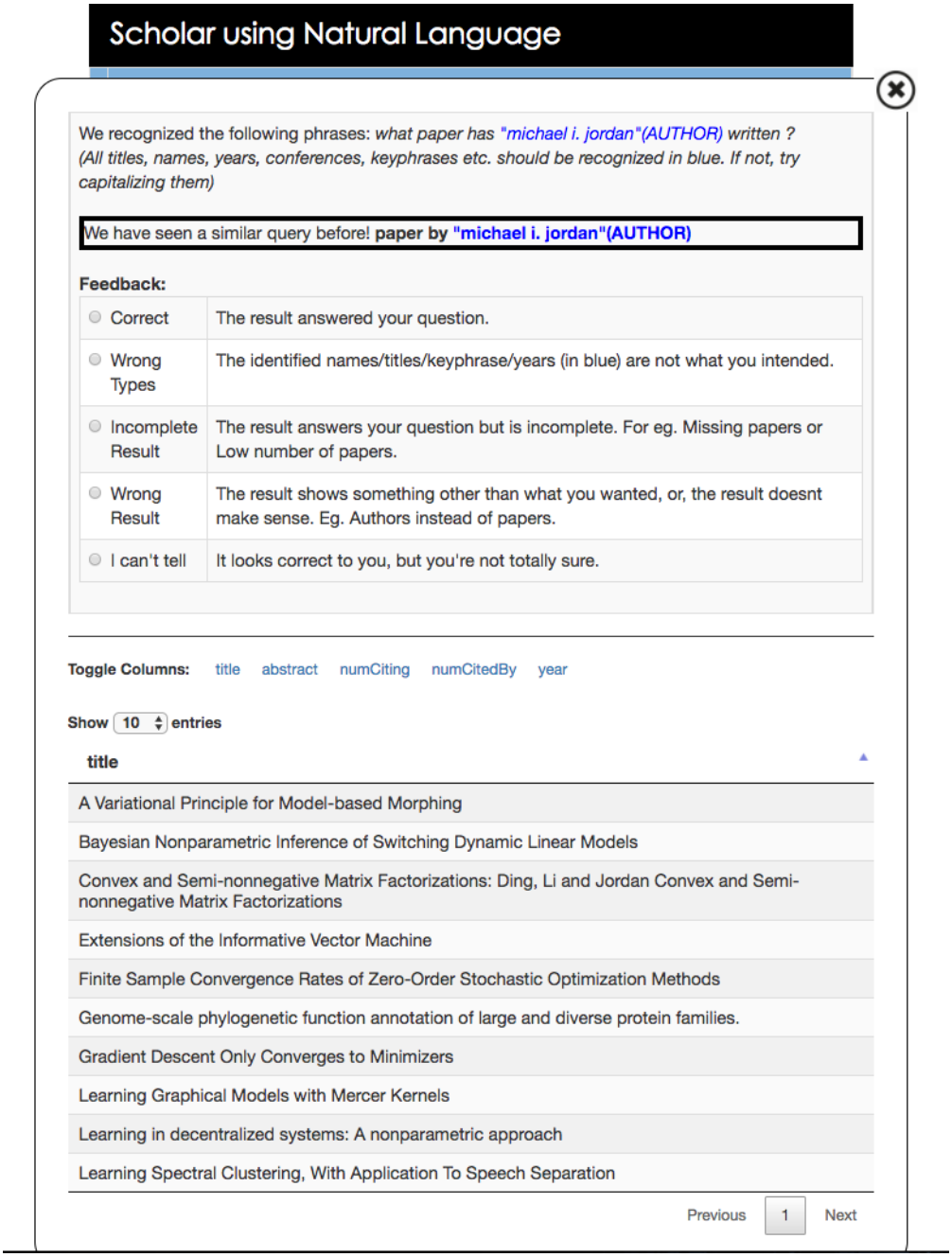

Figure 3.4: Once the user writes an utterance and pushes execute, they are presented with this screen. First, the identified entities and their types are highlighted to help them decide if the model is receiving the correct inputs. Second, in case the generated (anonymized) query already exists in the training set, it is presented as an alternate utterance, to give users additional confidence about the results that they are seeing. Third, they are presented with 5 feedback options as discussed in the main paper. Finally, they are presented with results and they can toggle columns with additional information.

Using these assists and the predicted results, users are asked to select from five feedback options: *Correct*, *Wrong Types*, *Incomplete Result*, *Wrong Result* and *Can't Tell*. The *Correct* and *Wrong Result* options represent scenarios when the user is satisfied with the result, or the result is identifiably wrong, respectively. *Wrong Types* indicates incorrect entity identification, which can be determined from type highlighting. *Incomplete Result* indicates that the query is correct but the result is not; this outcome can occur because the database is incomplete. *Can't Tell* indicates that the user is unsure about the feedback to provide.

3.6.2 Three-Stage Online Experiment

In this experiment, using our developed user interface, we use Algorithm 1 to learn a semantic parser from scratch. The experiment had three stages; in each stage, we recruited 10 new users (computer science graduate students) and asked them to issue at least 10 utterances each to the system and to provide feedback on the results. We considered results marked as either *Correct* or *Incomplete Result* as correct queries for learning. The remaining incorrect utterances were sent to a crowd worker for annotation and were used to retrain the system for the next stage. The crowd worker had prior experience in writing SQL queries and was hired from Upwork after completing a short SQL test. The worker was also given access to the database to be able to execute the queries and ensure that they are correct. For the first stage, the system was trained using 640 examples generated using templates, that were augmented to 1746 examples using paraphrasing (see Section 3.4.3). The complexity of the utterances issued in each of the three phases were comparable, in that, the average length of the correct SQL query for the utterances, and the number of tables required to be queried, were similar.

Table 3.5 shows the percent of utterances judged by users as either *Correct* or *Incomplete Result* in each stage. In the first stage, we do not have any labeled examples, and the model is trained using only synthetically generated data from schema templates and paraphrases (see Section 3.4.3). Despite the lack of real examples, the system correctly answers 25% of questions. The system's accuracy increases and annotation effort decreases in each successive stage as additional utterances are contributed and incorrect utterances are labeled. This result demonstrates that we can successfully build semantic parsers for new domains by using neural models to generate SQL with crowd-sourced annotations driven by user feedback.

We analyzed the feedback signals provided by the users in the final stage of the experiment to measure

	Stage 1	Stage 2	Stage 3
Accuracy (%)	25	53.7	63.5

Table 3.5: Percentage of utterances marked as *Correct* or *Incomplete* by users, in each stage of our online experiment.

	Feedback Error Rate (%)
Correct SQL	6.1
Incorrect SQL	6.3

Table 3.6: Error rates of user feedback when the SQL is correct and incorrect. The *Correct* and *Incomplete results* options are erroneous if the SQL query is correct, and vice versa for incorrect queries.

the quality of feedback. We found that 22.3% of the generated queries did not execute (and hence were incorrect). 6.1% of correctly generated queries were marked wrong by users (see Table 3.6). This erroneous feedback results in redundant annotation of already correct examples. The main cause of this erroneous feedback was incomplete data for aggregation queries, where users chose *Wrong* instead of *Incomplete*. 6.3% of incorrect queries were erroneously deemed correct by users. It is important that this fraction be low, as these queries become incorrectly-labeled examples in the training set that may contribute to the deterioration of model accuracy over time. This quality of feedback is already sufficient for our neural models to improve with usage, and creating better interfaces to make feedback more accurate is an important task for future work.

3.6.3 SCHOLAR dataset

We release a new semantic parsing dataset for academic database search using the utterances gathered in the user study. We augment these labeled utterances with additional utterances labeled by crowd workers. (Note that these additional utterances were not used in the online experiment). The final dataset comprises 816 natural language utterances labeled with SQL, divided into a 600/216 train/test split. We also provide a database on which to execute these queries containing academic papers with their authors, citations, journals, keywords and datasets used. Table 3.1 shows statistics of this dataset. Our parser achieves an accuracy of 67% on this train/test split in the fully supervised setting. In comparison, a nearest neighbor strategy that uses the cosine similarity metric using a TF-IDF representation for the utterances yields an accuracy of

52.75%.

We found that 15% of the predicted queries did not execute, predominantly owing to (1) accessing table columns without joining with those tables, and (2) generating incorrect types that could not be deanonymized using the utterance. The main types of errors in the remaining well-formed queries that produced incorrect results were (1) portions of the utterance (such as 'top' and 'cited by both') were ignored, and (2) some types from the utterance were not transferred to the SQL query.

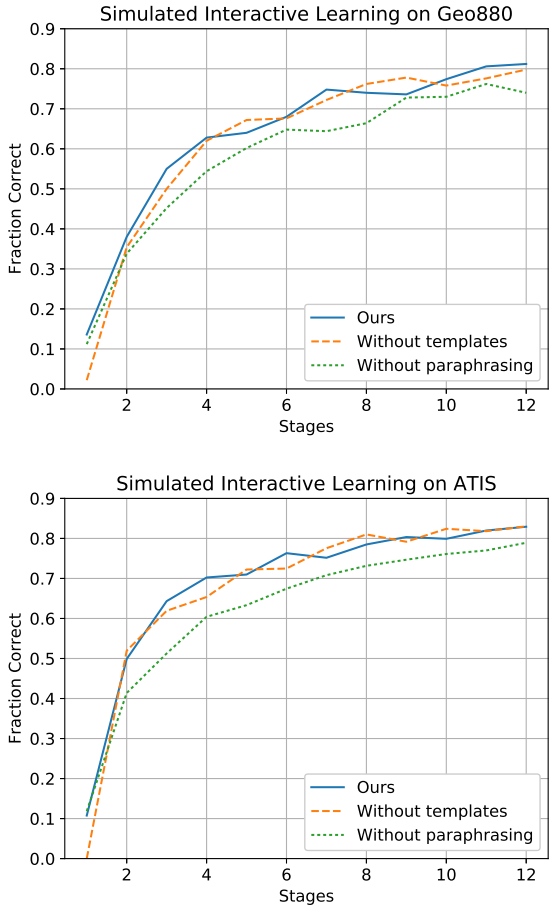

Figure 3.5: Accuracy as a function of batch number in simulated interactive learning experiments on Geo880 (top) and ATIS (bottom).

3.6.4 Simulated Interactive Experiments

We conducted additional simulated interactive learning experiments using GEO880 and ATIS to better understand the behavior of our train-deploy feedback loop, the effects of our data augmentation approaches,

Batch Size	150	100	50
% Wrong	70.2	60.4	54.3

Table 3.7: Percentage of examples that required annotation (i.e., where the model initially made an incorrect prediction) on GEO880 vs. batch size.

and the annotation effort required. We randomly divide each training set into K batches and present these batches sequentially to our interactive learning algorithm. Correctness feedback is provided by comparing the result of the predicted query to the gold query, i.e., we assume that users are able to perfectly distinguish correct results from incorrect ones.

Figure 3.5 shows accuracies on GEO880 and ATIS respectively of each batch when the model is trained on all previous batches. As in the live experiment, accuracy improves with successive batches. Data augmentation using templates helps in the initial stages of GEO880, but its advantage is reduced as more labeled data is obtained. Templates did not improve accuracy on ATIS, possibly because most ATIS queries involve two entities, i.e., a source city and a destination city, whereas our templates only generate questions with a single entity type. Nevertheless, templates are important in a live system to motivate users to interact with it in early stages. As observed before, paraphrasing improves performance at all stages.

Table 3.7 shows the percent of examples that require annotation using various batch sizes for GEO880. Smaller batch sizes reduce annotation effort, with a batch size of 50 requiring only 54.3% of the examples to be annotated. This result demonstrates that more frequent deployments of improved models leads to fewer mistakes.

3.7 Conclusion

In this Chapter, we describe an approach to rapidly train a semantic parser as a NLIDB for non-expert users, that iteratively improves parser accuracy over time while requiring minimal intervention. Our approach uses an attention-based neural sequence-to-sequence model, with data augmentation from the target database and paraphrasing, to parse utterances to SQL. This model is deployed in an online system, where user feedback on its predictions is used to select utterances to send for crowd worker annotation.

We find that the semantic parsing model is comparable in performance to previous systems that either

map from utterances to logical forms, or generate SQL, on two benchmark datasets, GEO880 and ATIS. We further demonstrate the effectiveness of our online system by learning a semantic parser from scratch for an academic domain. A key advantage of our approach is that it is not language-specific, and can easily be ported to other commonly used query languages, such as SPARQL or ElasticSearch. Finally, we also release a new dataset of utterances and SQL queries for an academic domain.

Chapter 4

Learning Programmatic Idioms for Scalable Semantic Parsing

Programmers typically organize executable source code using high-level coding patterns or idiomatic structures such as nested loops, exception handlers and recursive blocks, rather than as individual code tokens. In contrast, state of the art (SOTA) semantic parsers like the models we developed in Chapter 2, map natural language instructions to source code by building the code syntax tree one node at a time. In this Chapter, we introduce an iterative method to extract code idioms from large source code corpora by repeatedly collapsing most-frequent depth-2 subtrees of their syntax trees, and train semantic parsers to apply these idioms during decoding. Applying idiom-based decoding on the contextual source code generation task of Chapter 2 improves performance by 2.2% BLEU score while reducing training time by more than 50%. This improved speed enables us to scale up the model by training on an extended training set that is 5$\times$ larger, to further move up performance by an additional 2.3% BLEU and 0.9% exact match. Finally, idioms also significantly improve accuracy of semantic parsing to SQL on the ATIS-SQL dataset of Chapter 3, when training data is limited.

4.1 Introduction

When programmers translate Natural Language (NL) specifications into executable source code, they typically start with a high-level plan of the major structures required, such as nested loops, conditionals, etc. and then proceed to fill in specific details into these components. We refer to these high-level structures (Figure 4.1 (b)) as code idioms [Allamanis and Sutton, 2014]. In this chapter, we demonstrate how learning to use code idioms leads to an improvement in model accuracy and training time for the task of semantic parsing, i.e., mapping intents in NL into general purpose source code [Iyer et al., 2017; Ling et al., 2016].

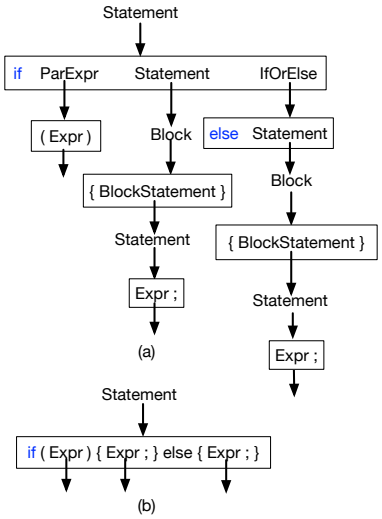

Figure 4.1: (a) Syntax tree based decoding for semantic parsing (for e.g. our model from Chapter 2) uses as many as 11 rules (steps) to produce the outer structure of a very frequently used `if-then-else` code snippet. (b) Direct application of an `if-then-else` idiom during decoding leads to improved accuracy and training time.

State-of-the-art semantic parsers are neural encoder-decoder models, where decoding is guided by the target programming language grammar [Yin and Neubig, 2017; Rabinovich et al., 2017; Iyer et al., 2018] to ensure syntactically valid programs, as we did in Chapter 2. For general purpose programming languages with large formal grammars, this can easily lead to long decoding paths even for short snippets of code. For example, Figure 4.1 shows an intermediate parse tree for a generic `if-then-else` code snippet, for which the decoder requires as many as eleven decoding steps before ultimately filling in the slots for the `if` condition, the `then` expression and the `else` expression. However, the `if-then-else` block can be seen as a higher level structure such as shown in Figure 4.1 (b) that can be applied in one decoding

step and reused in many different programs. In this dissertation, we refer to frequently recurring subtrees of programmatic parse trees as *code idioms*, and we equip semantic parsers with the ability to learn and directly generate idiomatic structures as in Figure 4.1 (b).

We introduce a simple iterative method to extract idioms from a dataset of programs by repeatedly collapsing the most frequent depth-2 subtrees of syntax parse trees. Analogous to the byte pair encoding (BPE) method [Gage, 1994; Sennrich et al., 2016] that creates new subtokens of words by repeatedly combining frequently occurring adjacent pairs of subtokens, our method takes a depth-2 syntax subtree and replaces it with a tree of depth-1 by removing all the internal nodes. This method is in contrast with the approach using probabilistic tree substitution grammars (pTSG) taken by Allamanis and Sutton [2014], who use the *explanation quality* of an idiom to prioritize idioms that are more interesting, with an end goal to suggest useful idioms to programmers using IDEs. Once idioms are extracted, we greedily apply them to semantic parsing training sets to provide supervision for learning to apply idioms.

We evaluate our approach on two semantic parsing tasks that map NL into 1) general-purpose source code (from Chapter 2), and 2) executable SQL queries (from Chapter 3), respectively. On the first task, i.e., context dependent semantic parsing using the CONCODE dataset, we improve model performance by 2.2% of BLEU score. Furthermore, generating source code using idioms results in a more than 50% reduction in the number of decoding steps, which cuts down training time to less than half, from 27 to 13 hours. Taking advantage of this reduced training time, we further push performance on CONCODE to an EM of 13.4 and a BLEU score of 28.9 by training on an extended version of the training set (with 5$\times$ the number of training examples). On the second task, i.e., mapping NL utterances into SQL queries for a flight information database (ATIS-SQL from Chapter 3), using idioms significantly improves denotational accuracy over SOTA models, when a limited amount of training data is used, and also marginally outperforms the SOTA when the full training set is used (more details in Section 4.7).

4.2 Related Work

Neural encoder-decoder models have proved effective in mapping NL to logical forms [Dong and Lapata, 2016] and also for directly producing general purpose programs (Chapters 2 and 3). Ling et al. [2016] use a sequence-to-sequence model with attention and a copy mechanism to generate source code. Instead of

directly generating a sequence of code tokens, recent methods focus on constrained decoding mechanisms to generate syntactically correct output using a decoder that is either grammar-aware or has a dynamically determined modular structure paralleling the structure of the abstract syntax tree (AST) of the code [Rabinovich et al., 2017; Krishnamurthy et al., 2017; Yin and Neubig, 2017]. In Chapter 2, we use a similar decoding approach but use a specialized context encoder for the task of context-dependent code generation. We augment these neural encoder-decoder models with the ability to decode in terms of frequently occurring higher level idiomatic structures to achieve gains in accuracy and training time.

Another different but related method to produce source code is using sketches, which are code snippets containing slots in the place of low-level information such as variable names, method arguments, and literals. Dong and Lapata [2018] generate such sketches using programming language-specific sketch creation rules and use them as intermediate representations to train token-based seq2seq models that convert NL to logical forms. Hayati et al. [2018] retrieve sketches from a large training corpus and modify them for the current input; Murali et al. [2018] use a combination of neural learning and type-guided combinatorial search to convert existing sketches into executable programs, whereas Nye et al. [2019] additionally also generate the sketches before synthesising programs. Our idiom-based decoder learns to produce commonly used subtrees of programming syntax-trees in one decoding step, where the non-terminal leaves function as slots that can be subsequently expanded in a grammer-aware fashion. Code idioms can be roughly viewed as a tree-structured generalization of sketches, that can be automatically extracted from large code corpora for any programming language, and unlike sketches, can also be nested with other idioms or grammar rules.

More closely related to the idioms that we use for decoding is Allamanis and Sutton [2014], who develop a system (HAGGIS) to automatically mine idioms from large code bases. They focus on finding interesting and explainable idioms, e.g., those that can be included as preset code templates in programming IDEs. Instead, we learn frequently used idioms that can be easily associated with NL phrases in our dataset. The production of large subtrees in a single step directly translates to a large speedup in training and inference.

Concurrent with our research, Shin et al. [2019] also develop a system (PATOIS) for idiom-based semantic parsing and demonstrate its benefits on the Hearthstone [Ling et al., 2016] and Spider [Yu et al., 2018b] datasets. While we extract idioms by collapsing frequently occurring depth-2 AST subtrees and apply them greedily during training, they use non-parametric Bayesian inference for idiom extraction and train neural

models to either apply entire idioms or generate its full body.

4.3 Idiom Aware Encoder-Decoder Models

We aim to train semantic parsers having the ability to use idioms during code generation. To do this, we first extract frequently used idioms from the training set, and then provide them as supervision to the semantic parser's learning algorithm.

Formally, if a semantic parser decoder is guided by a grammar $\mathscr{G} = (N, \Sigma, R)$, where N and Σ are the sets of non-terminals and terminals respectively, and R is the set of production rules of the form $A \rightarrow \beta, A \in N, \beta \in \{N \cup \Sigma\}^*$, we would like to construct an idiom set I with rules of the form $B \rightarrow \gamma, B \in N, \gamma \in \{N \cup \Sigma\}^*$, such that $B \overset{\geq 2}{\Longrightarrow} \gamma$ under $\mathscr{G}$, i.e., γ can be derived in two or more steps from B under $\mathscr{G}$. For the example in Figure 4.1, R would contain rules for expanding each non-terminal, such as *Statement* $\rightarrow$ *if ParExpr Statement IfOrElse* and *ParExpr* $\rightarrow$ *{ Expr }*, whereas I would contain the idiomatic rule *Statement* $\rightarrow$ *if (Expr) { Expr ; } else { Expr ; }* .

The decoder builds trees from $\hat{\mathscr{G}} = (N, \Sigma, R \cup I)$. Although the set of valid programs under both $\mathscr{G}$ and $\hat{\mathscr{G}}$ are exactly the same, this introduction of ambiguous rules into $\mathscr{G}$ in the form of idioms presents an opportunity to learn shorter derivations. In the next two sections, we describe the idiom extraction process, i.e., how I is chosen, and the idiom application process, i.e., how the decoder is trained to learn to apply idioms.

4.4 Idiom Extraction

Algorithm 2 describes the procedure to add idiomatic rules, I, to the regular production rules, R. Our goal is to populate the set I by identifying frequently occurring idioms (subtrees) from the programs in training set $\mathscr{D}$. Since enumerating all subtrees of every AST in the training set is infeasible, we observe that all subtrees s' of a frequently occurring subtree s are just as or more frequent than s, so we take a bottom-up approach by repeatedly collapsing the most frequent depth-2 subtrees. Intuitively, this can be viewed as a particular kind of generalization of the BPE [Gage, 1994; Sennrich et al., 2016] algorithm for sequences, where new subtokens are created by repeatedly combining frequently occurring adjacent pairs of subtokens. Note that

```
1  Procedure Extract-Idioms(𝒟, 𝒢, n)
       Input: 𝒟 → Training Programs
       Input: 𝒢 → Parsing Grammar
       Input: n → Number of idioms
2      T ← {} ▷ Stores all parse trees
3      for d ∈ 𝒟 do
4          T ← T ∪ Parse-Tree(d, 𝒢)
5      end
6      I ← {}
7      for i ← 1 to K do
8          s ← Most-frequent Depth-2-Subtree(T)
9          for t ∈ T do
10             t ← Collapse-Subtree(t, s)
11         end
12         I ← I ∪ {s}
13     end
14 end
15 Procedure Collapse-Subtree(t, s)
       Input: t → parse-tree
       Input: s → Depth 2 subtree
16     while s subtree-of t do
17         Replace s in t with Collapse(s)
18     end
19 end
20 Procedure Collapse(s)
       Input: s → Depth 2 subtree
21     frontier ← leaves(s)
22     return Tree(root(s),frontier)
23 end
```

Algorithm 2: Idiom Extraction.

subtrees of parse trees have an additional constraint, i.e., either all or none of the children of non-terminal nodes are included, since a grammar rule has to be used entirely or not at all.

We perform idiom extraction in an iterative fashion. We first populate T with all parse trees of programs in $\mathscr{D}$ using grammar $\mathscr{G}$ (Step 4). Each iteration then comprises retrieving the most frequent depth-2 subtree s from T (Step 8), followed by post-processing T to replace all occurrences of s in T with a collapsed (depth-1) version of s (Step 10 and 17). The collapse function (Step 20) simply takes a subtree, removes all its internal nodes and attaches its leaves directly to its root (Step 22). The collapsed version of s is a new idiomatic rule (a depth-1 subtree), which we add to our set of idioms, I (Step 12). We illustrate two iterations of this algorithm in Figure 4.2 ((a)-(b) and (c)-(d)). Assuming (a) is the most frequent depth-2

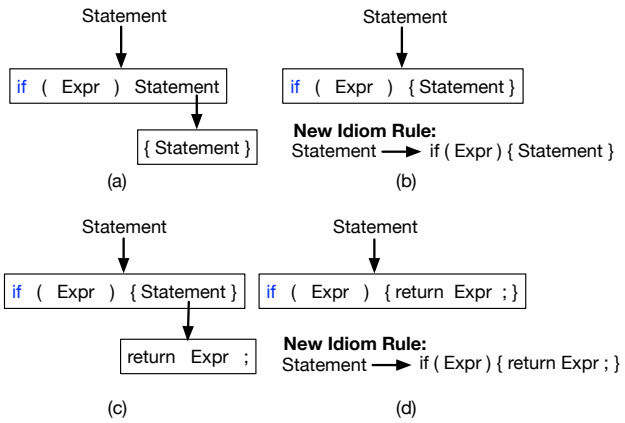

Figure 4.2: Two steps of the idiom extraction process described in Algorithm 2. (a) First, we find the most frequent depth-2 syntax subtree under Grammar $\mathscr{G}$ in dataset $\mathscr{D}$, collapse it to produce a new production rule (b), and replace all occurrences in the dataset with the collapsed version. Next, (c) is the most frequent depth-2 subtree which now happens to contain (b) within it. (c) is collapsed to form an idiom (d), which is effectively a depth-3 idiom.

subtree in the dataset, it is transformed into the idiomatic rule in (b). Larger idiomatic trees are learned by combining several depth-2 subtrees as the algorithm progresses. This is shown in Figure 4.2 (c) which contains the idiom extracted in (b) within it owing to the post-processing of the dataset after idiom (b) is extracted (Step 10 of Algorithm 2) which effectively makes the idiom in (d), a depth-3 idiom. We perform idiom extraction for K iterations. In our experiments we vary the value of K based on the number of idioms we would like to extract.

4.5 Model Training with Idioms

Once a set of idioms I is obtained, we next train semantic parsing models to apply these idioms while decoding. We do this by supervising grammar rule generation in the decoder using a compressed set of rules for each example, using the idiom set I (see Algorithm 3). More concretely, we first obtain the parse tree t_i (or grammar rule set p_i) for each training program y_i under grammar $\mathscr{G}$ (Step 3) and then greedily collapse each depth-2 subtree in t_i corresponding to every idiom in I (Step 5). Once t_i cannot be further collapsed, we translate t_i into production rules r_i based on the collapsed tree, with $|r_i| \leq |p_i|$ (Step 7).

This process is illustrated in Figure 4.3 where we perform two applications of the first idiom from Figure 4.2 (b), followed by one application of the second idiom from Figure 4.2 (d), after which the tree cannot be

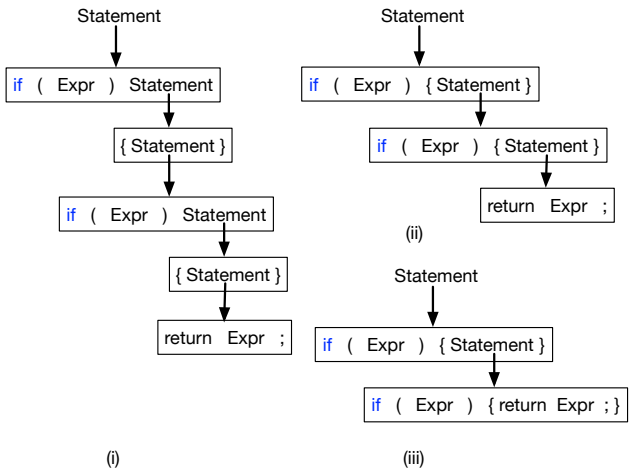

Figure 4.3: (i) Application of the two idioms extracted in Figure 4.2 on a new training example. (ii) We first perform two applications of the idiom in Figure 4.2b, (iii) followed by an application of the idiom in Figure 4.2d. The resulting tree can be represented with just 2 parsing rules instead of 5.

Procedure *Compress*($\mathscr{D}, \mathscr{G}, I$)
  Input: $\mathscr{D} \rightarrow$ Training Programs
  Input: $\mathscr{G} \rightarrow$ Parsing Grammar
  Input: $I \rightarrow$ Idiom set
  for $d \in \mathscr{D}$ **do**
    $t \leftarrow$ Parse-Tree$(d, \mathscr{G})$
    for $i \in I$ **do**
      $t \leftarrow$ Collapse-Subtree(t, i)
    end
    Train decoder using Production-Rules(t)
  end
end

Algorithm 3: Training Example Compression.

further compressed using those two idioms. The final tree can be represented using $|r_i| = 2$ rules instead of the original $|p_i| = 5$ rules. The decoder is then trained similar to previous approaches [Yin and Neubig, 2017] using the compressed set of rules. We observe a rule set compression of more than 50% in our experiments (Section 4.7).

4.6 Experimental Setup

We apply our approach on 1) the context dependent encoder-decoder model of Chapter 2 on the CONCODE dataset which we outperform, and 2) the task of mapping NL utterances to SQL queries on the ATIS-SQL

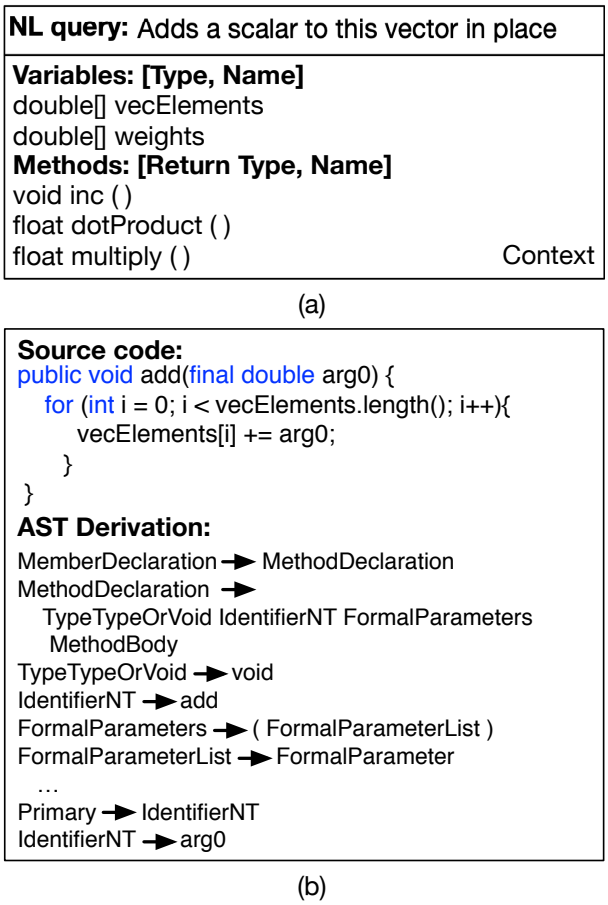

Figure 4.4: Context dependent code generation task of Chapter 2 that involves mapping an NL query together with a set of context variables and methods (each having an identifier name and data type) (a) into source code, represented as a sequence of grammar rules (b).

dataset [Iyer et al., 2017] of Chapter 3, where an idiom-based model using the full training set outperforms the SOTA, also achieving significant gains when using a reduced training set.

4.6.1 Context Dependent Semantic Parsing

The contextual code generation task introduced in Chapter 2 involves mapping an NL query together with a class context comprising a list of variables (with types) and methods (with return types), into the source code of a class member function. For the convenience of the reader, we briefly re-describe the task here. Figure 4.4 (a) shows an example where the context comprises variables and methods (with types) that would normally exist in a class that implements a vector, such as `vecElements` and `dotProduct()`. Conditioned on

this context, the task involves mapping the NL query *Adds a scalar to this vector in place* into a sequence of parsing rules to generate the source code in Figure 4.4 (b).

Formally, the task is: Given a NL utterance q, a set of context variables $\{v_i\}$ with types $\{t_i\}$, and a set of context methods $\{m_i\}$ with return types $\{r_i\}$, predict a set of parsing rules $\{a_i\}$ of the target program. The best performing model of Chapter 2 was a neural encoder-decoder with a context-aware encoder and a decoder that produces a sequence of Java grammar rules.

Baseline Model We first improve upon the model of Chapter 2 by making three major modifications in the encoder, which yields gains in both speed and accuracy (Iyer-Simp). We then use the resulting model as a baseline for testing the effectiveness of code idioms. First, in addition to camel-case splitting of identifier tokens, we use byte-pair encoding (BPE) [Sennrich et al., 2016] on all NL tokens, identifier names and types and embed all these BPE tokens using a single embedding matrix. Next, we replace the RNN that contextualizes the subtokens of identifiers and types with an average of the subtoken embeddings instead. Finally, we consolidate the three separate RNNs for contextualizing NL, variable names with types, and method names with types, into a single shared RNN, which greatly reduces the number of model parameters. Formally, let $\{q_i\}$ represent the set of BPE tokens of the NL, and $\{t_{ij}\}$, $\{v_{ij}\}$, $\{r_{ij}\}$ and $\{m_{ij}\}$ represent the jth BPE token of the ith variable type, variable name, method return type, and method name respectively. First, all these elements are embedded using a BPE token embedding matrix B to give us $\mathbf{q_i}$, $\mathbf{t_{ij}}$, $\mathbf{v_{ij}}$, $\mathbf{r_{ij}}$ and $\mathbf{m_{ij}}$. Using Bi-LSTM f, the encoder then computes:

$$h_1, \cdots, h_z = f(\mathbf{q_1}, \ldots, \mathbf{q_z}) \tag{4.1}$$

$$\mathbf{v_i} = Avg(\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}}) \tag{4.2}$$

$$\text{Similarly, compute } \mathbf{m_i}, \mathbf{t_i}, \mathbf{r_i} \tag{4.3}$$

$$\hat{t_i}, \hat{v_i} = f(\mathbf{t_i}, \mathbf{v_i}) \tag{4.4}$$

$$\hat{r_i}, \hat{m_i} = f(\mathbf{r_i}, \mathbf{m_i}) \tag{4.5}$$

Then, $h_1, \ldots, h_z$, and $\hat{t_i}, \hat{v_i}, \hat{r_i}, \hat{m_i}$ are passed to the attention mechanism in the decoder, exactly as in Chapter 2. The decoder remains the same as described in Chapter 2, and produces a probability distribution

```
List all flights from Denver to Seattle

SELECT DISTINCT flight_1.flight_id FROM
 flight f1, airport_service as1, city c1,
 airport_service as2, city c2 WHERE
 f1.from_airport = as1.airport_code
 AND as1.city_code = c1.city_code
 AND c1.city_name = "Denver"
 AND f1.to_airport = as2.airport_code
 AND as2.city_code = c2.city_code
 AND c2.city_name = "Seattle";
```

Figure 4.5: Example NL utterance with its corresponding executable SQL query from the ATIS-SQL dataset.

over grammar rules at each time step. This forms our baseline model (Iyer-Simp).

Idiom Aware Training To utilize idioms, we augment this decoder by retrieving the top-K most frequent idioms from the training set (Algorithm 2), followed by post-processing the training set by greedily applying these idioms (Algorithm 3; we denote this model as Iyer-Simp-K). We evaluate all our models on the CONCODE dataset which was created using Java source files from `github.com`. It contains 100K tuples of (NL, code, context) for training, 2K tuples for development, and 2K tuples for testing. We use a BPE vocabulary of 10K tokens (for matrix B) and get the best validation set results using the original hyperparameters from Chapter 2. Since idiom aware training is significantly faster than without idioms, it enables us to train on an additional 400K training examples from the full version of CONCODE (see Chapter 2). We report exact match accuracy, corpus level BLEU score (which serves as a measure of partial credit) [Papineni et al., 2002], and training time for all these configurations.

4.6.2 Semantic Parsing to SQL

This task, as discussed in Chapter 3, involves mapping NL utterances into executable SQL queries. We use the ATIS-SQL dataset (introduced in Chapter 3), that contains NL questions posed to a flight database, together with their SQL queries and a database with 25 tables to execute them against (Figure 4.5 shows an example). The dataset is split into 4,379 training, 491 validation, and 448 testing examples following Kwiatkowski et al. [2011].

The SOTA is a Seq2Seq model with attention (described in Chapter 3) and achieves a denotational

accuracy of 82.5% on the test set. Since using our idiom-based approach requires a model that uses grammar-rule based decoding, we use a modified version of the Seq2Prod model described in Chapter 2 (based on Yin and Neubig [2017]) as a baseline model (Seq2Prod), and augment its decoder with SQL idioms (Seq2Prod-K).

Seq2Prod is an encoder-decoder model, where the encoder executes an n-layer bi-LSTM over NL embeddings and passes the final layer LSTM hidden states to an attention mechanism in the decoder. Note that the Seq2Prod encoder described in Chapter 2 encodes a concatenated sequence of NL and context, but ATIS-SQL instances do not include contextual information. Thus, if q_i represents each lemmatized token of the NL, they are first embedded using a token embedding matrix B to give us $\mathbf{q_i}$. Using Bi-LSTM f, the encoder then computes:

$$h_1, \cdots, h_z = f(\mathbf{q_1}, \ldots, \mathbf{q_z}) \tag{4.6}$$

Then, $h_1, \ldots, h_z$ are passed to the attention mechanism in the decoder.

Similar to the Iyer-Simp model described above, the sequential decoder uses attention and produces a sequence of grammar rules $\{a_t\}$(a_t at step t), which can later be put together to form a source code snippet. As before, the decoder is an LSTM-based RNN with hidden dimension size H, that produces a context vector c_t at each time step, which is used to compute a distribution over next actions.

$$p(a_t|a_{<t}) \propto exp(W^{n_t} c_t) \tag{4.7}$$

Here, W^{n_t} is a $|n_t| \times H$ matrix, where $|n_t|$ is the total number of unique grammar rules that n_t can be expanded to. The context vector c_t is computed using the hidden state s_t of an n-layer decoder LSTM cell and attention vectors over the NL z_t, as described below.

Decoder LSTM The decoder uses an n-layer LSTM (LSTM$_f$) whose hidden state at time t, s_t, is computed based on an embedding of the current non-terminal n_t to be expanded, a contextualized embedding of the previous production rule a_{t-1}, a contextualized embedding of the parent production rule, par(n_t), that produced n_t, and the previous decoder LSTM state s_{t-1}. Note that unlike the previous Iyer-Simp model and

unlike the Seq2Prod model of Chapter 2, we do not use the parent LSTM state as it does not provide any improved performance.

We use an embedding matrix N to embed n_t. Note that, instead of using direct embeddings of rules a_{t-1} and par(n_t) in LSTM$_f$ as done by the previous Iyer-Simp model and the Seq2Prod model of Chapter 2, we use contextualized embeddings. To compute the contextualized embeddings of a_{t-1} and par(n_t) in LSTM$_f$, we use another single layer Bi-LSTM (LSTM$_g$) across the left and right sides of the rule (using separator symbol `SEP`) and use the final hidden state as inputs to LSTM$_f$ instead. More concretely, if a grammar rule is represented as $A \rightarrow B_1 \dots B_n$, then:

$$\text{Emb}(A \rightarrow B_1 \dots B_n) = \text{LSTM}_g(\{A, \texttt{SEP}, B_1, \dots, B_n\}) \tag{4.8}$$

$$s_t = \text{LSTM}_f(n_t, \text{Emb}(a_{t-1}), \text{Emb}(\text{par}(n_t)), s_{t-1}) \tag{4.9}$$

This modification can help the LSTM$_f$ cell locate the position of n_t within rules a_{t-1} and par(n_t), especially, for lengthy idiomatic rules. s_t is then used for single-step attention.

Single-Step Attention At time step t, the decoder attends to every token in the NL representation, h_i, using the current decoder state, s_t, to compute a set of attention weights α_t, which are used to combine h_i into an NL context vector z_t. We use the general attention mechanism of Luong et al. [2015].

$$\alpha_{t,i} = \frac{\exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}{\sum_i \exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}$$

$$z_t = \sum_i \alpha_{t,i} h_i$$

Finally, c_t is computed using the decoder state and context vector z_t:

$$c_t = tanh(\hat{W}[s_t : z_t])$$

Supervised Copy Mechanism The supervised copy mechanism is exactly the same as described for the previous model (Iyer-Simp), using context vector c_t.

Model	Exact	BLEU
Seq2Seq†	3.2 (2.9)	23.5 (21.0)
Seq2Prod†	6.7 (5.6)	21.3 (20.6)
Iyer et al. [2018] (from Chapter 2)†	8.6 (7.1)	22.1 (21.3)
Iyer-Simp	12.5 (9.8)	24.4 (23.2)
Iyer-Simp + 200 idioms	12.2 (9.8)	**26.6 (24.0)**

Table 4.1: Exact Match and BLEU scores for our simplified model (Iyer-Simp) with and without idioms, compared with results from Chapter 2† on the test (validation) set of CONCODE. Iyer-Simp achieves significantly better EM and BLEU score and reduces training time from 40 hours to 27 hours. Augmenting the decoding process with 200 code idioms further pushes up BLEU and reduces training time to 13 hours.

Hyperparameters We use an embedding size H of 1024 for NL query tokens. Both the encoder and decoder LSTM cells use 2-layers and a hidden dimensionality of 1024 (512 on each direction for BiLSTMs). We use an embedding size of 512 for encoding non-terminals and a hidden size of 256 for the contextualized rules (for LSTM$_g$) in the decoder. We use dropout with $p = 0.5$ in between LSTM layers and at the output of the decoder over c_t. We train our model for 60 epochs using mini-batch gradient descent with a batch size of 40, and we use Adam [Kingma and Ba, 2015] with an initial learning rate of 0.001 for optimization. We decay our learning rate by 80% based on performance on the development set after every epoch. We use beam search with a beam size of 5 for decoding the sequence of grammar rules at test time.

Idiom Aware Training As before, we augment the set of decoder grammar rules with top-K idioms extracted from ATIS-SQL. To represent SQL queries as grammar rules, we use the python `sqlparse` package.

4.7 Results and Discussion

Table 4.1 presents exact match and BLEU scores on the original CONCODE train/validation/test split of Chapter 2. Iyer-Simp yields a large improvement of 3.9 EM and 2.2 BLEU over the best model of Chapter 2, while also being significantly faster (27 hours for 30 training epochs as compared to 40 hours). Using a reduced BPE vocabulary makes the model memory efficient, which allows us to use a larger batch size that in turn speeds up training. Furthermore, using 200 code idioms further improves BLEU by 2.2% while maintaining comparable EM accuracy. Using the top-200 idioms results in a target AST compression of

Model	Exact	BLEU	Training Time (h)
Seq2Seq†	2.9	21.0	12
Seq2Prod†	5.6	20.6	36
Iyer et al. [2018] (from Chapter 2)†	7.1	21.3	40
Iyer-Simp	9.8	23.2	27
+ 100 idioms	9.8	24.5	15
+ 200 idioms	9.8	24.0	13
+ 300 idioms	9.6	23.8	12
+ 400 idioms	9.7	23.8	11
+ 600 idioms	9.9	22.7	11

Table 4.2: Variation in Exact Match, BLEU score, and training time on the validation set of CONCODE with number of idioms used. After top-200 idioms are used, accuracies start to reduce, since using more specialized idioms can hurt model generalization. Training time plateaus after considering top-600 idioms.

Model	Exact	BLEU
1× Train	12.0 (9.7)	26.3 (23.8)
2× Train	13.0 (10.3)	28.4 (25.2)
3× Train	13.3 (10.4)	28.6 (26.5)
5× Train	13.4 (11.0)	28.9 (26.6)

Table 4.3: Exact Match and BLEU scores on the test (validation) set of CONCODE by training Iyer-Simp-400 on additional data from the full extended training set from Chapter 2. Significant improvements in training speed after incorporating idioms makes training on large amounts of data possible.

more than 50%, which results in fewer decoder RNN steps being performed. This reduces training time further by more than 50%, from 27 hours to 13 hours.

In Table 4.2, we illustrate the variations in EM, BLEU and training time with the number of idioms. We find that 200 idioms performs best overall in terms of balancing accuracy and training time. Adding more idioms continues to reduce training time, but accuracy also suffers. Since we permit idioms to contain identifier names to capture frequently used library methods, having too many idioms hurts generalization, especially since the test set is built using repositories disjoint from the training set. Finally, the amount of compression, and therefore the training time, plateaus after the top-600 idioms are incorporated.

Compared to the model of Chapter 2, our significantly reduced training time enables us to train on additional examples from the full extended training set (see Chapter 2). We run Iyer-Simp using 400 idioms (taking advantage of even lower training time) on up to 5 times the amount of data, while making sure that

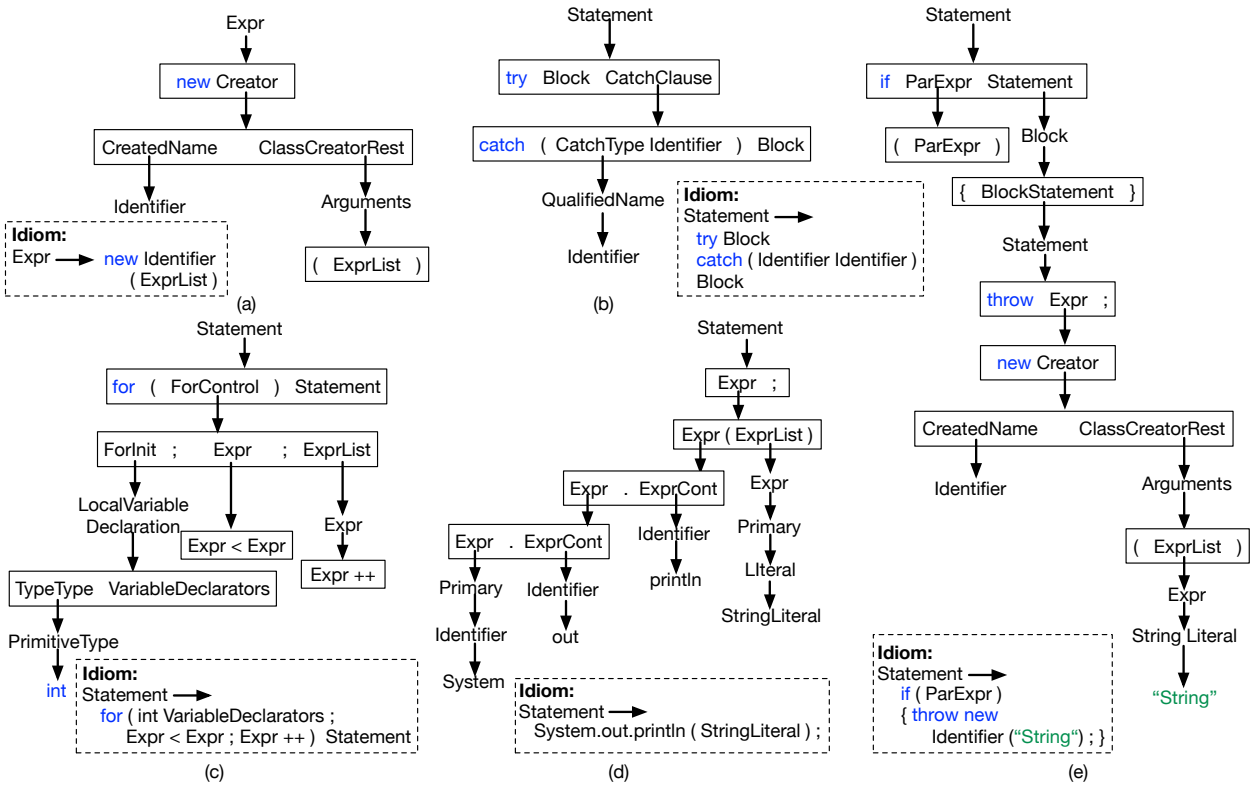

Figure 4.6: Examples of idioms learned from CONCODE. (a)-(c) represent idioms for instantiation of a new object, exception handling and integer-based looping respectively. (d) represents an idiom for applying a very commonly used library method (`System.out.println`). (e) is a combination of various idioms viz. an if-then idiom, an idiom for throwing exceptions and finally, reusing idiom (a) for creating new objects.

we do not include in training any NL from the validation or the test sets. Since the original set of idioms learned from the original training set are quite general, we directly use them rather than relearn the idioms from scratch. We report EM and BLEU scores for different amounts of training data on the same validation and test sets as CONCODE in Table 4.3. In general, accuracies increase with the amount of data with the best model achieving a BLEU score of 28.9 and EM of 13.4.

Figure 4.6 shows example idioms extracted from CONCODE: (a) is an idiom to construct a new object with arguments, (b) represents a `try-catch` block, and, (c) is an integer-based `for` loop. In (e), we show how small idioms are combined to form larger ones; it combines an `if-then` idiom with a `throw-exception` idiom, which throws an object instantiated using idiom (a). The decoder also learns idioms to directly generate common library methods such as `System.out.println( StringLiteral )` in

Model	Accuracy
Iyer et al. [2017]†	82.5
Seq2Prod	79.1
Seq2Prod + 400 idioms	**83.2**

Table 4.4: Denotational Accuracy for Seq2Prod with and without idioms, compared with results from Chapter 3† on the test set of ATIS using SQL queries. Results averaged over 3 runs.

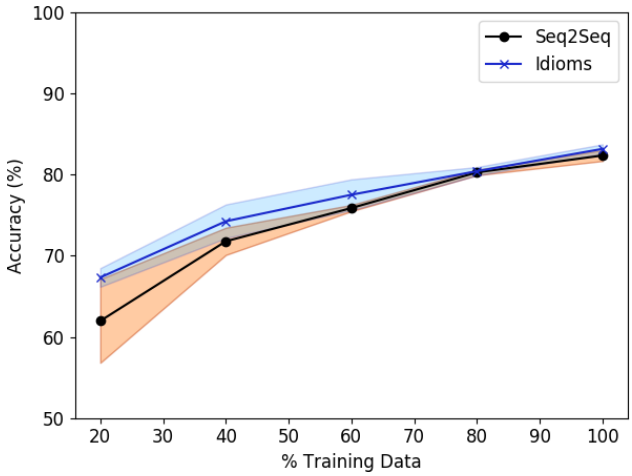

Figure 4.7: Accuracy of the Seq2Seq baseline (from Chapter 3) compared with Seq2Prod-K, on the test set of ATIS-SQL for varying fractions of training data. Idioms significantly boost accuracy when training data is scarce. (Mean and Std. Dev. computed across 3 runs). 20-40% use 100 idioms and the rest use 400 idioms.

one decoding step (d).

For the NL to SQL task, we report denotational accuracy in Table 4.4. We observe that Seq2Prod underperforms the Seq2Seq model of Chapter 3, most likely because a SQL query parse is much longer than the original query. This is remedied by using top-400 idioms, which compresses the decoded sequence size, marginally outperforming the SOTA (83.2%). Finegan-Dollak et al. [2018] observed that the SQL structures in ATIS-SQL are repeated numerous times in both train and test sets, thus facilitating Seq2seq models to memorize these structures without explicit idiom supervision. To test a scenario with limited repetition of structures, we compare Seq2Seq with Seq2Prod-K for limited training data (increments of 20%) and observe that (Figure 4.7) idioms are additionally helpful with lesser training data, consistent with our intuition.

4.8 Conclusions

We presented a general approach to make semantic parsers aware of target idiomatic structures, by first identifying frequently used idioms, followed by providing models with supervision to apply these idioms. We demonstrated this approach on the context dependent code generation task of Chapter 2 where we achieved a new SOTA in EM accuracy and BLEU score. We also found that decoding using idioms significantly reduces training time and allows us to train on significantly larger datasets. Finally, our approach also outperformed the SOTA for the semantic parsing to SQL task of Chapter 3 on ATIS-SQL, with significant improvements under a limited training data regime.

Chapter 5

Summarizing Source Code using a Neural Attention Model

High quality source code is often paired with high level summaries of the computation it performs, for example in code documentation or in descriptions posted in online forums. Such summaries are extremely useful for applications such as code search, but are expensive to manually author, hence only done for a small fraction of all code that is produced. Also, if such summaries could be automatically generated, they would be invaluable in NL to code systems to explain model predictions back to users in order to improve trust. In this Chapter, we present the first completely data-driven approach for generating high level summaries of source code. Our model, CODE-NN, uses Long Short Term Memory (LSTM) networks with attention to produce sentences that describe C# code snippets and SQL queries. CODE-NN is trained on a new corpus that is automatically collected from `stackoverflow.com`, which we release. Experiments demonstrate strong performance on two tasks: (1) code summarization, where we establish the first end-to-end learning results and outperform strong baselines, and (2) code retrieval, where our learned model improves the state of the art on a recently introduced C# benchmark by a large margin.

1. Source Code (C#):

```
public int TextWidth(string text) {
  TextBlock t = new TextBlock();
  t.Text = text;
  return (int)Math.Ceiling(t.ActualWidth);
}
```

Descriptions:

a. Get rendered width of string rounded up to the nearest integer
b. Compute the actual textwidth inside a textblock

2. Source Code (C#):

```
  var input = "Hello";
  var regEx = new Regex("World");
  return !regEx.IsMatch(input);
```

Descriptions:

a. Return if the input doesn't contain a particular word in it
b. Lookup a substring in a string using regex

3. Source Code (SQL):

```
SELECT Max(marks) FROM stud_records WHERE marks <
  (SELECT Max(marks) FROM stud_records);
```

Descriptions:

a. Get the second largest value of a column
b. Retrieve the next max record in a table

Figure 5.1: Code snippets in C# and SQL and their summaries in NL, from StackOverflow. Our goal is to automatically generate summaries from code snippets.

5.1 Introduction

Billions of lines of source code reside in online repositories [Dyer et al., 2013], and high quality code is often coupled with natural language (NL) in the form of instructions, comments, and documentation. Short summaries of the overall computation the code performs provide a particularly useful form of documentation for a range of applications, such as code search or tutorials. However, such summaries are expensive to manually author. As a result, this laborious process is only done for a small fraction of all code that is produced. If such summaries could be automatically generated, such a system would not only be useful to generate descriptions for existing undocumented code but would also be invaluable in NL to code systems to explain model predictions back to users in order to improve trust.

In this Chapter, we present the first completely data-driven approach for generating short high-level summaries of source code snippets in natural language. We focus on C#, a general-purpose imperative

language, and SQL, a declarative language for querying databases. Figure 5.1 shows example code snippets with descriptions that summarize the overall function of the code, with the goal to generate high level descriptions, such as *lookup a substring in a string*.

Generating such a summary is often challenging because the text can include complex, non-local aspects of the code (e.g., consider the phrase 'second largest' in Example 3 in Figure 5.1). In addition to being directly useful for interpreting uncommented code or automatically generated code, high-quality generation models can also be used for code retrieval, and in turn, for natural language programming by applying nearest neighbor techniques to a large corpus of automatically summarized code.

Natural language generation has traditionally been addressed as a pipeline of modules that decide 'what to say' (content selection) and 'how to say it' (realization) separately [Reiter and Dale, 2000; Wong and Mooney, 2007; Chen et al., 2010; Lu and Ng, 2011]. Such approaches require supervision at each stage and do not scale well to large domains. We instead propose an end-to-end neural network called CODE-NN that jointly performs content selection using an attention mechanism, and surface realization using Long Short Term Memory (LSTM) networks. The system generates a summary one word at a time, guided by an attention mechanism over embeddings of the source code, and by context from previously generated words provided by a LSTM network [Hochreiter and Schmidhuber, 1997]. The simplicity of the model allows it to be learned from the training data without the burden of feature engineering [Angeli et al., 2010] or the use of an expensive approximate decoding algorithm [Konstas and Lapata, 2013].

Our model is trained on a new dataset of code snippets with short descriptions, created using data gathered from `http://stackoverflow.com`, a popular programming help website. Since access is open and unrestricted, the content is inherently noisy (ungrammatical, non-parsable, lacking content), but as we will see, it still provides strong signal for learning. To reliably evaluate our model, we also collect a clean, human-annotated test set.[1]

We evaluate CODE-NN on two tasks: code summarization and code retrieval (Section 5.2). For summarization, we evaluate using automatic metrics such as METEOR and BLEU-4, together with a human study for naturalness and informativeness of the output. The results show that CODE-NN outperforms a number of strong baselines and, to the best of our knowledge, CODE-NN is the first approach that learns to generate

[1] Data and code are available at `https://github.com/sriniiyer/codenn`.

summaries of source code from easily gathered online data. We further use CODE-NN for code retrieval for programming related questions on a recent C# benchmark, and results show that CODE-NN improves the state of the art (Allamanis et al. [2015b]) for mean reciprocal rank (MRR) by a wide margin.

5.2 Tasks

CODE-NN generates a NL summary of source code snippets (GEN task). We have also used CODE-NN on the inverse task to retrieve source code given a question in NL (RET task).

Formally, let U_C be the set of all code snippets and U_N be the set of all summaries in NL. For a training corpus with J code snippet and summary pairs $(c_j, n_j), 1 \leq j \leq J, c_j \in U_C, n_j \in U_N$, we define the following two tasks:

GEN For a given code snippet $c \in U_C$, the goal is to produce a NL sentence $n^* \in U_N$ that maximizes some scoring function $s \in (U_C \times U_N \rightarrow \mathbb{R})$:

$$n^* = \underset{n}{\operatorname{argmax}}\ s(c, n) \tag{5.1}$$

RET We also use the scoring function s to retrieve the highest scoring code snippet c_j^* from our training corpus, given a NL question $n \in U_N$:

$$c_j^* = \underset{c_j}{\operatorname{argmax}}\ s(c_j, n), 1 \leq j \leq J \tag{5.2}$$

In this work, s is computed using an LSTM neural attention model, to be described in Section 5.5.

5.3 Related Work

Although we focus on generating high-level summaries of source code snippets, there has been work on producing code descriptions at other levels of abstraction. Movshovitz-Attias and Cohen [2013] study the task of predicting class-level comments by learning n-gram and topic models from open source Java projects and testing it using a character-saving metric on existing comments. Allamanis et al. [2015a] create models

for suggesting method and class names by embedding them in a high dimensional continuous space. Sridhara et al. [2010] present a pipeline that generates summaries of Java methods by selecting relevant content and generating phrases using templates to describe them. There is also work on improving program comprehension [Haiduc et al., 2010], identifying cross-cutting source code concerns [Rastkar et al., 2011], and summarizing software bug reports [Rastkar et al., 2010]. To the best of our knowledge, we are the first to use learning techniques to construct completely new sentences from arbitrary code snippets.

Source code summarization is also related to generation from formal meaning representations. Wong and Mooney [2007] present a system that learns to generate sentences from lambda calculus expressions by inverting a semantic parser. Mei et al. [2016], Konstas and Lapata [2013], and Angeli et al. [2010] create learning algorithms for text generation from database records, again assuming data that pairs sentences with formal meaning representations. In contrast, we present algorithms for learning from easily gathered web data.

In the database community, Simitsis and Ioannidis [2009] recognize the need for SQL database systems to talk back to users. Koutrika et al. [2010] built an interactive system (LOGOS) that translates SQL queries to text using NL templates and database schemas. Similarly there has been work on translating SPARQL queries to natural language using rules to create dependency trees for each section of the query, followed by a transformation step to make the output more natural [Ngonga Ngomo et al., 2013]. These approaches are not learning based, and require significant manual template-engineering efforts.

We use recurrent neural networks (RNN) based on LSTMs and neural attention to jointly model source code and NL. Recently, RNN-based approaches have gained popularity for text generation and have been used in machine translation [Sutskever et al., 2011], image and video description [Karpathy and Li, 2015; Venugopalan et al., 2015; Devlin et al., 2015], sentence summarization [Rush et al., 2015], and Chinese poetry generation [Zhang and Lapata, 2014]. Perhaps most closely related, Wen et al. [2015] generate text for spoken dialogue systems with a two-stage approach, comprising an LSTM decoder semantically conditioned on the logical representation of speech acts, and a reranker to generate the final output. In contrast, we design an end-to-end attention-based model for source code.

For code retrieval, Allamanis et al. [2015b] proposed a system that uses Stackoverflow data and web search logs to create models for retrieving C# code snippets given NL questions and vice versa. They

construct distributional representations of code structure and language and combine them using additive and multiplicative models to score (code, language) pairs, an approach that could work well for retrieval but cannot be used for generation. We learn a neural generation model without using search logs and show that it can also be used to score code for retrieval, with much higher accuracy.

Synthesizing code from language is an alternative to code retrieval and has been studied in both the Systems and NLP research communities. Giordani and Moschitti [2012], Li and Jagadish [2014], and Gulwani and Marron [2014] synthesize source code from NL queries for database and spreadsheet applications. Similarly, Lei et al. [2013] interpret NL instructions to machine-executable code, and Kushman and Barzilay [2013] convert language to regular expressions. Unlike most synthesis methods, CODE-NN is language agnostic, as we demonstrate its applications on both C# and SQL.

5.4 Dataset

We collected data from StackOverflow (SO), a popular website for posting programming-related questions. Anonymized versions of all the posts can be freely downloaded.[2] Each post can have multiple tags. Using the *C#* tag for C# and the *sql*, *database* and *oracle* tags for SQL, we were able to collect 934,464 and 977,623 posts respectively.[3] Each post comprises a short title, a detailed question, and one or more responses, of which one can be marked as accepted. We found that the text in the question and responses is domain-specific and verbose, mixed with details that are irrelevant for our tasks. Also, code snippets in responses that were not accepted were frequently incorrect or tangential to the question asked. Thus, we extracted only the title from the post and use the code snippet from those accepted answers that contain exactly one code snippet (using `<code>` tags). We add the resulting (title, query) pairs to our corpus, resulting in a total of 145,841 pairs for C# and 41,340 pairs for SQL.

Cleaning We train a semi-supervised classifier to filter titles like *'Difficult C# if then logic'* or *'How can I make this query easier to write?'* that bear no relation to the corresponding code snippet. To do so, we annotate 100 titles as being *clean* or *not clean* for each language and use them to bootstrap the algorithm. We then use the remaining titles in our training set as an unsupervised signal, and obtain a classification accuracy

[2] `http://archive.org/details/stackexchange`

[3] The data was downloaded in Dec 2014.

of over 73% on a manually labeled test set for both languages. For the final dataset, we retain 66,015 C# (title, query) pairs and 32,337 SQL pairs that are classified as clean, and use 80% of these datasets for training, 10% for validation and 10% for testing.

Parsing Given the informal nature of StackOverflow, the code snippets are approximate answers that are usually incomplete. For example, we observe that only 12% of the SQL queries parse without any syntactic errors (using zql[4]). We therefore aim to perform a best-effort parse of the code snippet, using modified versions of an ANTLR parser for C# [Parr, 2013] and *python-sqlparse* [Albrecht, 2015] for SQL. We strip out all comments and to avoid being context specific, we replace literals with tokens denoting their types. In addition, for SQL, we replace table and column names with numbered placeholder tokens while preserving any dependencies in the query. For example, the SQL query in Figure 5.1 is represented as `SELECT MAX(col0) FROM tab0 WHERE col0 < (SELECT MAX(col0) FROM tab0).`

Data Statistics The structural complexity and size of the code snippets in our dataset makes our tasks challenging. More than 40% of our C# corpus comprises snippets with three or more statements and functions, and 20% contains loops and conditionals. Also, over a third of our SQL queries contain one or more subqueries and multiple tables, columns and functions (like `MIN`, `MAX`, `SUM`). On average, our C# snippets are 38 tokens long and the queries in our corpus are 46 tokens long, while titles are 9-12 words long. Table 5.2 shows the complete data statistics.

Human Annotation For the GEN task, we use n-gram based metrics (see Section 5.6.1) of the summary generated by our model with respect to the actual title in our corpus. Titles can be short, and a given code snippet can be described in many different ways with little overlapping content between them. For example, the descriptions for the second code snippet in Figure 5.1 share very few words with each other. To address these limitations, we extend our test set by asking human annotators to provide two additional titles for 200 snippets chosen at random from the test set, making a total of three reference titles for each code snippet. To collect this data, annotators were shown only the code snippets and were asked to write a short summary after looking at a few example summaries. They were also asked to "think of a question that they could ask on a programming help website, to get the code snippet as a response." This encouraged them to briefly

[4] `http://zql.sourceforge.net`

C#	**# Statements**		**# Functions**	
	≥ 3	23,611 (44.7%)	≥ 3	26,541 (51.0%)
	≥ 4	17,822 (33.7%)	≥ 4	20,221 (38.2%)
	# Loops		**# Conditionals**	
	≥ 1	10,676 (20.0%)	≥ 1	11,819 (22.3%)
SQL	**# Subqueries**		**# Tables**	
	≥ 1	11,418 (35%)	≥ 3	14,695 (44%)
	≥ 2	3,625 (11%)	≥ 4	10,377 (31%)
	# Columns		**# Functions**	
	≥ 5	12,366 (37%)	≥ 3	6,290 (19%)
	≥ 6	9,050 (27%)	≥ 4	3,973 (12%)

Table 5.1: Statistics for code snippets in our dataset.

C#	**Avg. code length**	38 tokens	**# tokens**	91,156
	Avg. title length	12 words	**# words**	24,857
SQL	**Avg. query length**	46 tokens	**# tokens**	1,287
	Avg. title length	9 words	**# words**	10,086

Table 5.2: Average code and title lengths together with vocabulary sizes for C# and SQL after post-processing.

describe the key feature that the code is trying to demonstrate. We use half of this test set for model tuning (DEV, see Section 5.5) and the rest for evaluation (EVAL).

5.5 The CODE-NN Model

Description We present an end-to-end generation system that performs content selection and surface realization jointly. Our approach uses an attention-based neural network to model the conditional distribution of a NL summary n given a code snippet c. Specifically, we use an LSTM model that is guided by attention on the source code snippet to generate a summary one word at a time, as shown in Figure 5.2.[5]

Formally, we represent a NL summary $n = n_1, \ldots, n_l$ as a sequence of 1-hot vectors $\mathbf{n_1}, \ldots, \mathbf{n_l} \in \{0,1\}^{|N|}$, where N is the vocabulary of the summaries. Our model computes the probability of n (scoring function s in Eq. 5.1) as a product of the conditional next-word probabilities

$$s(c, n) = \prod_{i=1}^{l} p(n_i | n_1, \ldots, n_{i-1})$$

[5]We experimented with other sequence [Sutskever et al., 2014] and tree based architectures [Tai et al., 2015] as well. None of these models significantly improved performance, however, this is an important area for future work.

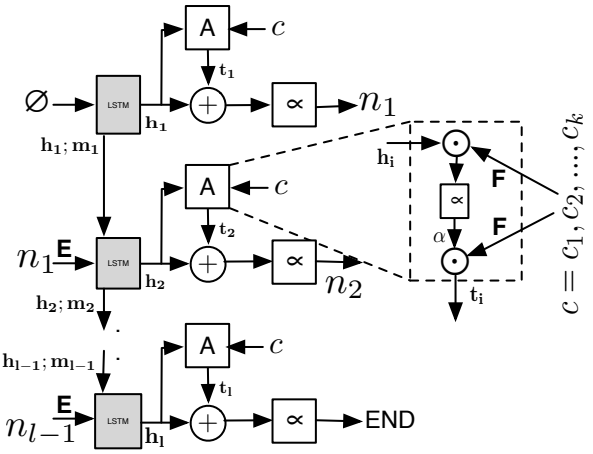

Figure 5.2: Generation of a title $n = n_1, \ldots,$ END given code snippet $c_1, ..., c_k$. The attention cell computes a distributional representation t_i of the code snippet based on the current LSTM hidden state h_i. A combination of t_i and h_i is used to generate the next word, n_i, which feeds back into the next LSTM cell. This is repeated until a fixed number of words or END is generated. $\propto$ blocks denote softmax operations.

with,

$$p(n_i|n_1, \ldots, n_{i-1}) \propto \mathbf{W} \tanh(\mathbf{W_1 h_i} + \mathbf{W_2 t_i})$$

where, $\mathbf{W} \in \mathbb{R}^{|N| \times H}$ and $\mathbf{W_1}, \mathbf{W_2} \in \mathbb{R}^{H \times H}$, H being the embedding dimensionality of the summaries. $\mathbf{t_i}$ is the contribution from the attention model on the source code (see below). $\mathbf{h_i}$ represents the hidden state of the LSTM cell at the current time step and is computed based on the previously generated word, the previous LSTM cell state $\mathbf{m_{i-1}}$ and the previous LSTM hidden state $\mathbf{h_{i-1}}$ as

$$\mathbf{m_i}; \mathbf{h_i} = f(\mathbf{n_{i-1}E}, \mathbf{m_{i-1}}, \mathbf{h_{i-1}}; \theta)$$

where $\mathbf{E} \in \mathbb{R}^{|N| \times H}$ is a word embedding matrix for the summaries. We compute f using the LSTM cell architecture used by Zaremba et al. [Zaremba and Sutskever, 2014].

Attention The generation of each word is guided by a global attention model [Luong et al., 2015], which computes a weighted sum of the embeddings of the code snippet tokens based on the current LSTM state (see right part in Figure 5.2). Formally, we represent c as a set of 1-hot vectors $\mathbf{c_1}, \ldots, \mathbf{c_k} \in \{0, 1\}^{|C|}$ for each source code token; C is the vocabulary of all tokens in our code snippets. Our attention model

computes,

$$\mathbf{t_i} = \sum_{j=1}^{k} \alpha_{i,j} \cdot \mathbf{c_j F}$$

where $\mathbf{F} \in \mathbb{R}^{|C| \times H}$ is a token embedding matrix and each $\alpha_{i,j}$ is proportional to the dot product between the current internal LSTM hidden state $\mathbf{h_i}$ and the corresponding token embedding $\mathbf{c_j}$:

$$\alpha_{i,j} = \frac{\exp(\mathbf{h_i}^{\mathrm{T}} \mathbf{c_j F})}{\sum_{j=1}^{k} \exp(\mathbf{h_i}^{\mathrm{T}} \mathbf{c_j F})}$$

Training We perform supervised end-to-end training using backpropagation [Werbos, 1990] to learn the parameters of the embedding matrices $\mathbf{F}$ and $\mathbf{E}$, transformation matrices $\mathbf{W}$, $\mathbf{W_1}$ and $\mathbf{W_2}$, and parameters θ of the LSTM cell that computes f. We use multiple epochs of minibatch stochastic gradient descent and update all parameters to minimize the negative log likelihood (NLL) of our training set. To prevent over-fitting we make use of dropout layers [Srivastava et al., 2014] at the summary embeddings and the output softmax layer. Using pre-trained embeddings [Mikolov et al., 2013a] for the summary embedding matrix or adding additional LSTM layers did not improve performance for the GEN task. Since the NLL training objective does not directly optimize for our evaluation metric (METEOR), we compute METEOR (see Section 5.6.1) on a small development set (DEV) after every epoch and save the intermediate model that gives the maximum score, as the final model.

Decoding Given a trained model and an input code snippet c, finding the most optimal title entails generating the title n^* that maximizes $s(c, n)$ (see Eq. 5.1). We approximate n^* by performing beam search on the space of all possible summaries using the model output.

Implementation Details We add special START and END tokens to our training sequences and replace all tokens and output words occurring with a frequency of less than 3 with an UNK token, making $|C| = 31,667$ and $|N| = 7,470$ for C# and $|C| = 747$ and $|N| = 2,506$ for SQL. Our hyper-parameters are set based on performance on the validation set. We use a minibatch size of 100 and set the dimensionality of the LSTM hidden states, token embeddings, and summary embeddings (H) to 400. We initialize all model parameters

uniformly between -0.35 and 0.35. We start with a learning rate of 0.5 and start decaying it by a factor of 0.8 after 60 epochs if accuracy on the validation set goes down, and terminate training when the learning rate goes below 0.001. We cap the parameter gradients to 5 and use a dropout rate of 0.5.

We use the Torch framework[6] to train our models on GPUs. Training runs for about 80 epochs and takes approximately 7 hours. We compute METEOR score at every epoch on the development set (DEV) to choose the best final model, with the best results obtained between 60 and 70 epochs. For decoding, we set the beam size to 10, and the maximum summary length to 20 words.

5.6 Experimental Setup

5.6.1 GEN Task

Baselines

For the GEN task, we compare CODE-NN with a number of competitive systems, none of which had been previously applied to generate text from source code, and hence we adapt them slightly for this task, as explained below.

IR is an information retrieval baseline that outputs the title associated with the code c_j in the training set that is closest to the input code c in terms of token Levenshtein distance. In this case s from Eq.5.1 becomes,

$$s(c, n_j) = -1 \times \text{lev}(c_j, c), 1 \leq j \leq J$$

MOSES [Koehn et al., 2007] is a popular phrase-based machine translation system. We perform generation by treating the tokenized code snippet as the source language, and the title as the target. We train a 3-gram language model using KenLM [Heafield, 2011] to use with MOSES, and perform MIRA-based tuning [Cherry and Foster, 2012] of hyper-parameters using DEV.

SUM-NN is the neural attention-based abstractive summarization model of Rush et al. [Rush et al., 2015]. It uses an encoder-decoder architecture with an attention mechanism based on a fixed context window of previously generated words. The decoder is a feed-forward neural language model that generates the next

[6] `http://torch.ch`

word based on previous words in a context window of size k. In contrast, we decode using an LSTM network that can model long range dependencies and our attention weights are tied to the LSTM hidden states. We set the embedding and hidden state dimensions and context window size by tuning on our validation set. We found this model to generate overly short titles like *'sql server 2008'* when a length restriction was not imposed on the output text. Therefore, we fix the output length to be the average title length in the training set while decoding.

Evaluation Metrics

We evaluate the GEN task using automatic metrics, and also perform a human study.

Automatic Evaluation We report METEOR [Banerjee and Lavie, 2005] and sentence level BLEU-4 [Papineni et al., 2002] scores. METEOR is recall-oriented and measures how well our model captures content from the references in our output. BLEU-4 measures the average n-gram precision on a set of reference sentences, with a penalty for overly short sentences. Since the generated summaries are short and there are multiple alternate summaries for a given code snippet, higher order n-grams may not overlap. We remedy this problem by using +1 smoothing [Lin and Och, 2004]. We compute these metrics on the tuning set DEV and the held-out evaluation set EVAL.

Human Evaluation Since automatic metrics do not always agree with the actual quality of the results [Stent et al., 2005], we perform human evaluation studies to measure the output of our system and baselines across two modalities, namely naturalness and informativeness. For the former, we asked 5 native English speakers to rate each title against grammaticality and fluency, on a scale between 1 and 5. For *informativeness* (i.e., the amount of content carried over from the input code to the NL summary, ignoring fluency of the text), we asked 5 human evaluators familiar with C# and SQL to evaluate the system output by rating the factual overlap of the summary with the reference titles, on a scale between 1 and 5.

5.6.2 RET task

Model and Baselines

CODE-NN As described in Section 5.2, for a given NL question n in the RET task, we rank all code snippets c_j in our corpus by computing the scoring function $s(c_j, n)$, and return the query c_j^* that maximizes it (Eq. 5.2).

RET-IR is an information retrieval baseline that ranks the candidate code snippets using cosine similarity between the given NL question n and all summaries n_j in the retrieval set, based on their vector representations using TF-IDF weights over unigrams. The scoring function s in Eq. 5.2 becomes:

$$s(c_j, n) = \frac{\text{tf-idf}(n_j) \cdot \text{tf-idf}(n)}{\|\text{tf-idf}(n_j)\|\|\text{tf-idf}(n)\|}, 1 \leq j \leq J$$

Evaluation Metrics

We assess ranking quality by computing the Mean Reciprocal Rank (MRR) of c_j^*. For every snippet c_j in EVAL (and DEV), we use two of the three references (title and human annotation), namely $n_{j,1}, n_{j,2}$. We then build a retrieval set comprising $(c_j, n_{j,1})$ together with 49 random distractor pairs $(c', n'), c' \neq c_j$ from the test set. Using $n_{j,2}$ as the natural language question, we rank all 50 items in this retrieval set and use the rank of query c_j^* to compute MRR. We average MRR over all returned queries c_j^* in the test set, and repeat this experiment for several different random sets of distractors.

5.6.3 Tasks from Allamanis et al. [2015b]

Allamanis et al. [2015b] take a retrieval approach to answer C# related natural language questions (L to C), similar to our RET task. In addition, they also use retrieval to summarize C# source code (C to L) and evaluate both tasks using the MRR metric. Although they also use data from Stackoverflow, their dataset preparation and cleaning methods differs significantly from ours. For example, they filter out posts where the question has fewer than 2 votes, the answer has fewer than 3 votes, or the post has fewer than 1000 views. Additionally, they also filter code snippets that cannot be parsed by Roslyn (.NET compiler) or are longer than 300 characters. Thus, to directly compare with their model, we re-train our generation model on

	Model	METEOR	BLEU-4
C#	IR	7.9 (6.1)	13.7 (12.6)
	MOSES	9.1 (9.7)	11.6 (11.5)
	SUM-NN	10.6 (10.3)	19.3 (18.2)
	CODE-NN	**12.3 (13.4)**	**20.5 (20.4)**
SQL	IR	6.3 (8.0)	13.5 (13.0)
	MOSES	8.3 (9.7)	15.4 (15.9)
	SUM-NN	6.4 (8.7)	13.3 (14.2)
	CODE-NN	**10.9 (14.0)**	**18.4 (17.0)**

Table 5.3: Performance on EVAL (DEV) for the GEN task.

	Model	Naturalness	Informativeness
C#	IR	3.42	2.25
	MOSES	1.41	2.42
	SUM-NN	**4.61***	1.99
	CODE-NN	**4.48**	**2.83**
SQL	IR	3.21	2.58
	MOSES	2.80	2.54
	SUM-NN	4.44	2.75
	CODE-NN	**4.54**	**3.12**

Table 5.4: Naturalness and Informativeness measures of model outputs. Stat. sig. between CODE-NN and others is computed with a 2-tailed Student's t-test; $p < 0.05$ except for *.

their dataset and use our model score for retrieval of both code and summaries.

5.7 Results

5.7.1 GEN Task

Table 5.3 shows automatic evaluation metrics for our model and baselines. CODE-NN outperforms all the other methods in terms of METEOR and BLEU-4 score. We attribute this to its ability to perform better content selection, focusing on the more salient parts of the code by using its attention mechanism jointly with its LSTM memory cells. The neural models have better performance on C# than SQL. This is in part because, unlike SQL, C# code contains informative intermediate variable names that are directly related to the objective of the code. On the other hand, SQL is more challenging in that it only has a handful of keywords and functions, and summarization models need to rely on other structural aspects of the code.

Informativeness and naturalness scores for each model from our human evaluation study are presented in Table 5.4. In general, CODE-NN performs well across both dimensions. Its superior performance in terms

	Model	MRR
C#	RET-IR	0.42 ± 0.02 (0.44 ± 0.01)
	CODE-NN	$\mathbf{0.58 \pm 0.01}$ ($\mathbf{0.66 \pm 0.02}$)
SQL	RET-IR	0.28 ± 0.01 (0.4 ± 0.01)
	CODE-NN	$\mathbf{0.44 \pm 0.01}$ ($\mathbf{0.54 \pm 0.02}$)

Table 5.5: MRR for the RET task. Dev set results in parentheses.

	Model	MRR
L to C	Allamanis	0.182 ± 0.009
	CODE-NN	$\mathbf{0.590 \pm 0.044}$
C to L	Allamanis	0.434 ± 0.003
	CODE-NN	$\mathbf{0.461 \pm 0.046}$

Table 5.6: MRR values for the Language to Code (L to C) and the Code to Language (C to L) tasks using the C# dataset of Allamanis et al. [2015b]

of informativeness further supports our claim that it manages to select content more effectively. Although SUM-NN performs similar to CODE-NN on naturalness, its output lacks content and has very little variation (see Section 5.7.4), which also explains its surprisingly low score on informativeness.

5.7.2 RET Task

Table 5.5 shows the MRR on the RET task for CODE-NN and RET-IR, averaged over 20 runs for C# and SQL. CODE-NN outperforms the baseline by about 16% for C# and SQL. RET-IR can only output code snippets that are annotated with NL as potential matches. On the other hand, CODE-NN can rank even unannotated code snippets and nominate them as potential candidates. Hence, it can leverage vast amounts of such code available in online repositories like Github. To speed up retrieval when using CODE-NN, it could be one of the later stages in a multi-stage retrieval system and candidates may also be ranked in parallel.

5.7.3 Comparison with Allamanis et al.

We train CODE-NN on their dataset and evaluate using the same MRR testing framework (see Table 5.6). Our model performs significantly better for the Language to Code task (L to C) and slightly better for the Code to Language task (C to L). The attention mechanism together with the LSTM network is able to

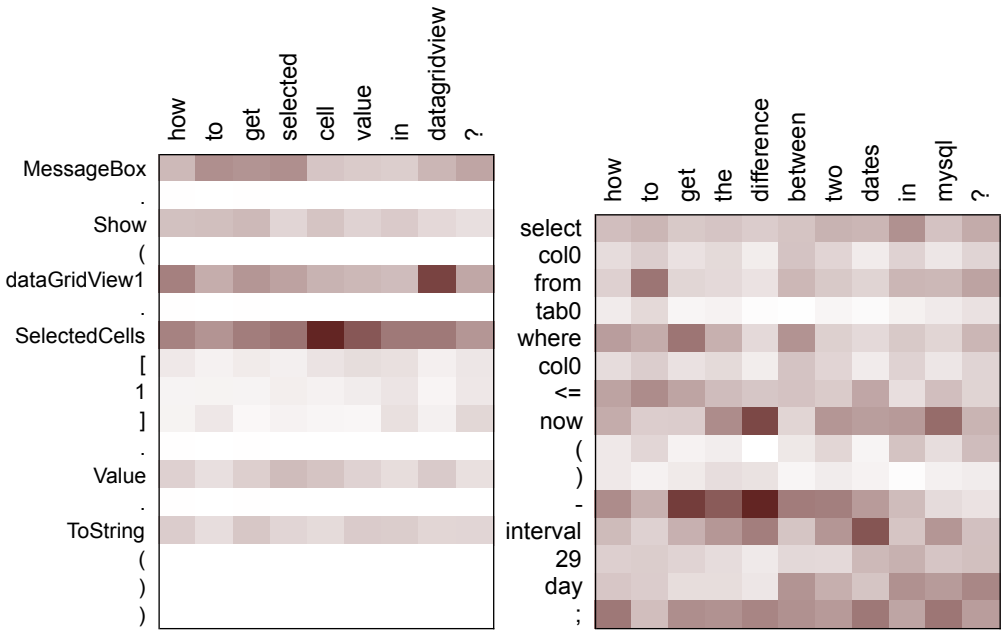

Figure 5.3: Heatmap of attention weights $\alpha_{i,j}$ for example C# (left) and SQL (right) code snippets. The model learns to align key summary words (like cell) with the corresponding tokens in the input (SelectedCells).

generate better scores for (language, code) pairs.

5.7.4 Qualitative Analysis

Figure 5.3 shows the relative magnitudes of the attention weights ($\alpha_{i,j}$) for example C# and SQL code snippets while generating their corresponding summaries. Darker regions represent stronger weights. CODE-NN automatically learns to do high-quality content selection by aligning key summary words with informative tokens in the code snippet.

Table 5.8 shows examples of the output generated by our model and baselines for code snippets in DEV. Most of the models produce meaningful output for simple code snippets (first example) but degrade on longer, compositional inputs. For example, the last SQL query listed in Table 5.8 includes a subquery, where a complete description should include both summing and concatenation. CODE-NN describes the summation (but not concatenation), while others return non-relevant descriptions.

Finally, we performed manual error analysis on 50 randomly selected examples from DEV (Table 5.7) for each language. Redundancy is a major source of error, i.e., generation of extraneous content-bearing

Error	% Cases
Correct	37%
Redundancy	17%
Missing Info	26%
Out of context	20%

Table 5.7: Error analysis on 50 examples in DEV

phrases, along with missing content, e.g., in the last example of Table 5.8 there is no reference to the concatenation operations present in the beginning of the query. Sometimes the output from our model can be out of context, in the sense that it does not match the input code. This often happens for low frequency tokens (7% of cases), for which CODE-NN realizes them with generic phrases. This also happens when there are very long range dependencies or compositional structures in the input, such as nested queries (13% of the cases).

5.8 Conclusion

In this Chapter, we presented CODE-NN, an end-to-end neural attention model using LSTMs to generate summaries of C# and SQL code by learning from noisy online programming websites. Such models are invaluable in NL to code systems to explain automatically generated source code back to users to ascertain them about the output being presented to them. Our model outperforms competitive baselines and achieves state of the art performance on automatic metrics, namely METEOR and BLEU, as well as on a human evaluation study. We also used CODE-NN to answer programming questions by retrieving the most appropriate code snippets from a corpus, and beat previous baselines for this task in terms of MRR. We have published our C# and SQL datasets, the accompanying human annotated test sets, and our code for the tasks described in this Chapter.

Method	Output
C# code	`var x = "FundList[10].Amount";` `int xIndex = Convert.ToInt32(Regex.Match(x,@"\d+").Value);`
Gold	Identify the number in given string
IR	Convert string number to integer
MOSES	How to xIndex numbers in C#?
SUM-NN	How can I get the value of a string?
CODE-NN	How to convert string to int?

C# code	`foreach (string pTxt in xml.parent) {`
	`  TreeNode parent = new TreeNode();`
	`  foreach (string cTxt in xml.child) {`
	`  TreeNode child = new TreeNode();`
	`  parent.Nodes.Add(child);`
	`  }`
	`}`

Gold	Adding childs to a treenode dynamically in C#
IR	How to set the name of a tabPage programmatically
MOSES	How can TreeView nodes from XML parentText string to a treeview node
SUM-NN	How to get data from xml file in C#
CODE-NN	How to get all child nodes in TreeView?

C# code	`string url = baseUrl + "/api/Entry/SendEmail?emailId=" + emailId;`
	`WebRequest req = WebRequest.Create(url);`
	`req.Method = "GET";`
	`req.BeginGetResponse(null, null);`

Gold	Execute a get request on a web server and receive the response asynchronously
IR	How to download a file from another Sharepoint Domain
MOSES	How baseUrl emailId C# how to a page in BeginGetResponse to
SUM-NN	How to get data from a file in C#
CODE-NN	How to call a URL from a web api post ?
SQL Query	`SELECT * FROM table ORDER BY Rand() LIMIT 10`
Gold	Select random rows from mysql table
IR	How to select a random record from a mysql database?
MOSES	How to select all records in mysql ?
SUM-NN	How can I select random rows from a table
CODE-NN	How to get random rows from a mysql database?

SQL Query	`SELECT Group_concat( Concat_ws(',', playerid, r1, r2) SEPARATOR ';')`
	`FROM (SELECT playerid, Sum(rank = 1) r1, Sum(rank < 5) r2`
	`  FROM result GROUP BY playerid) t;`

Gold	Get sum of group values based on condition and concatenate them into a string
IR	Mysql: counting occurences in a table, return as a single row
MOSES	Mysql query to get this result of the result of one column value in mysql
SUM-NN	How do i combine these two queries into one?
CODE-NN	How to get the sum of a column in a single query?

Table 5.8: Examples of outputs generated by each model for code snippets in DEV

Chapter 6

Conclusion

This dissertation presents efficient deep learning models and training paradigms for the task of mapping natural language to general purpose programming languages, which enables numerous applications for developers as well as non-expert users.

Chapter 2 was geared towards contextual source code generation for developers, where we presented deep learning methods to deal with modeling challenges relating to the significant dependence of the target code on the existing code context written by the developer. Additionally, we were able to learn models from inexpensive data gathered from large scale online code corpora and test them for generalization on held out code repositories. As part of this work, we also released the large CONCODE dataset to the community for progress on this contextual task. While we train a single encoder-decoder model, Guo et al. [2019] later present an alternative approach that combines a retrieval model and a meta-learner, where the former learns to find similar datapoints from the training data, and the latter considers retrieved datapoints as a pseudo task for fast adaptation.

Complementary to the notion of programming context for source code generation, subsequent research on tasks related to mapping NL to SQL across different database schemas [Yu et al., 2018b] treat the input database schema as contextual information, and condition the model output on encoded schema elements [Zhang et al., 2019]. Another way to incorporate such contextual information into models is to make the decoder grammar schema-dependent and then constrain the output based on the input schema. This direction is explored by Lin et al. [2019]. In addition to contextual knowledge, NL to code models also need to be

aware of background commonsense knowledge that are not part of the input, to understand the meaning of input phrases unseen during training. Dasigi [2019] make this distinction and also present methods to encode inputs together with the relevant background and real-world knowledge required to successfully generate the appropriate target program.

In Chapter 3, we presented methods to build extensible natural language interfaces to query arbitrary databases from scratch, in an inexpensive manner. These interfaces were equipped with the ability to improve over time as they see more user utterances. We tackled the shortcomings relating to annotation costs of earlier models, that map NL to specialized intermediate meaning representations, by training neural seq2seq models to map NL directly to a general purpose database query language (SQL). We showed that using user feedback to selectively annotate only incorrectly predicted SQL queries can reduce annotation costs, and that periodically retraining models using this feedback signal together with new annotations can result in a model that rapidly improves in performance over time. This work makes contributions to our broader goal of enabling regular developers of applications, software, and devices, to train NL interfaces for their products inexpensively.

The first step in our training paradigm was to deploy models that have been trained on synthetic parallel NL and SQL data, with the hypothesis that users will only successfully interact with models that are somewhat functional. The quality of the initial synthetic data thus directly impacts interface adoption, and therefore, impacts subsequent training stages. Basik et al. [2018] directly address this by developing DPPal, which uses various novel methods to produce a more effective initial synthetic dataset, and therefore, attains a significantly higher zero-shot performance on new database schemas.

In Chapter 4, we motivated the need to generate source code in terms of programmatic idioms, as a general technique that would make all NL to source code models train significantly faster, while also improving code generation accuracy. We presented a simple, yet effective method for extracting interpretable idioms from code datasets by collapsing the most frequently occurring depth-2 subtrees in their abstract syntax trees. We also presented a simple greedy strategy to incorporate these idioms into existing NL to code models without any changes to the models. Using idioms, we improved model performance for both developer-oriented contextual code generation, as well as for non-expert user-oriented semantic parsing, and were also able to take advantage of the faster training speed to train on more data in the same amount of

time, thus, significantly scaling up models to larger datasets.

Finally, in Chapter 5, we trained neural LSTM-based attentional models to generate descriptions of source code in NL, relating to their functionality, using datasets gathered from `stackoverflow.com`. Not only are these models useful for automatic code documentation and for summarization of source code snippets for improved searching, but they also represent an important component in NL to code systems that serves to assure users that the functionality that they desire is indeed the functionality provided by the generated source code.

While we only used source code tokens to produce code descriptions, the use of additional structural information has subsequently been shown to improve performance. For example, Alon et al. [2018] improve captioning performance on our dataset by representing the input code snippets as a set of compositional paths from its abstract syntax tree and using attention to select relevant paths while decoding. Similarly, LeClair et al. [2019] improve performance on alternate datasets by also encoding the AST of the input program. Our work has subsequently also inspired follow-up tasks such as generating code commit messages from code diffs using neural machine translation models [Jiang et al., 2017; Hu et al., 2018; Loyola et al., 2017]. Various new datasets have been created along similar lines to the ones we have presented in this paper from `stackoverflow.com`. While we only use SO examples with a single code snippet in their accepted answer, Yao et al. [2018] present a method to also leverage those posts containing multiple code snippets, by classifying each snippet as being standalone or not. Yin et al. [2018] create aligned NL-code datasets (CoNaLa) from `stackoverflow.com` with expanded coverage and also train specialized classifiers to filter the dataset to only high-quality examples.

We also show in Chapter 5 that description generating models can be effectively inverted to retrieve code snippets relevant to a NL instruction, from a large training corpus. Chen and Zhou [2018] improve retrieval performance on our dataset by jointly training two variational autoencoders (VAEs) to model bimodal source code and natural language data. Subsequent to this, Sachdev et al. [2018] take an alternative approach, where they train models to represent both NL and code snippets as continuous vectors in the same space. This enables faster retrieval, by using standard cosine similarity distances, and by the storage of precomputed code representations for the entire training corpus. Since documentation is almost always followed by code search, Yao et al. [2019] present methods to generate code summaries with the end goal of being later

advantageous for code retrieval.

In conclusion, this dissertation takes important steps towards the development of inexpensive ways to train efficient, robust and extensible machine learning based systems for mapping NL to general purpose source code. Allamanis et al. [2018] present a survey of various recent probabilistic methods that model the associations between NL and source code. In the process, they have also highlighted salient modeling differences between natural languages and programming languages, and their work serves as an excellent resource that complements the material in this thesis.

6.1 Future Work

In this section, we present ideas for extending our methods and tasks to map NL to general purpose source code. We present potential research directions relating to modeling improvements, such as scaling, using pretrained representations, and performing semantically constrained decoding, as well as relating to tasks, such as context dependent explanations, learning from background knowledge, and interactive programming.

Scalability While early datasets such as JOBS, GEO880 and ATIS, that were used for studying the task of mapping language to meaning representations/source code only contained a few thousand training examples, the popularity and inexpensive nature of learning from weak supervision in the last few years has resulted in the emergence of new datasets containing hundreds of thousands of examples. For example, in Chapter 4, we scale up our language to code models to a dataset with $500,000$ examples extracted from large online source code repositories without requiring any human annotations, and taking approximately 3 days to train. While this represents significant progress, online code repositories such as `github.com` still contain tens of millions of additional training examples that are still unused for training owing to the lack of fast and distributed training methods for our models. An important area for future work is to develop model architectures that can efficiently leverage both data and model parallelism, as well as large batch training [Ott et al., 2018], to take advantage of the orders of magnitude of additional source code data available online, in order to train significantly stronger NL to code models.

```
String getTitle (String html) {
    doc = Jsoup.parse(html);
    return doc.title();
}
```

Figure 6.1: A syntax-aware decoder can violate semantic constraints. The `doc` variable is used without being first declared.

Pretrained Representations for Source Code Similar to words in natural language, elements on the source code side such as API calls and programmatic idioms are related in terms of the contexts and applications that they are typically used for, and similarly, source code elements from different programming languages are also related. This presents an opportunity to learn representations for these elements using self-supervision [Peters et al., 2018; Devlin et al., 2019], that can be reused in various downstream NL to code tasks with a large potential of improving performance, particularly in low resource scenarios. The large datasets that we create and release as part of this dissertation can function as effective training sets to learn such representations and could be an important direction for future work.

Semantic Constraints and Source Code Invariants While we used syntax-based decoding [Yin and Neubig, 2017] in Chapter 2 to ensure the syntactic correctness of the generated code, the generated code can still violate various semantic constraints. For example, variables can still be used without being declared (Figure 6.1), types can be assigned to mismatching containers, and methods can be called on the wrong objects. Along similar lines, the decoder of Chapter 2 is unaware of programming invariants such as those related to the ordering of predicates in conjunctive clauses etc. Lin et al. [2019] address some of these constraints for SQL generation by recognizing that the grammar that produces tables and columns should respect the database schema, and by constraining the grammar to generate valid schema elements. An area for subsequent research is to determine ways to incorporate such semantic constraints for multiple programming languages in neural source code decoders while avoiding hard-coding an array of language-specific rules.

Context and Interaction-dependent Explanations In Chapter 5, we discussed methods to caption source code based on its functionality, ultimately serving as a means to explain generated source code back to

`delete_device_usage_data(**kwargs)`

When this action is called for a specified shared device, it allows authorized users to delete the device's entire previous history of voice input data and associated response data. This action can be called once every 24 hours for a specific shared device.

Request Syntax

```
{
   "DeviceArn": "string",
   "DeviceUsageType": "string"
}
```

Request Parameters

For information about the parameters that are common to all actions, see Common Parameters.
The request accepts the following data in JSON format.

DeviceArn

The ARN of the device.
Type: String
Pattern: `arn:[a-z0-9-\.]{1,63}:[a-z0-9-\.]{0,63}:[a-z0-9-\.]{0,63}:[a-z0-9-\.]{0,63}:[^/].{0,1023}`
Required: Yes

DeviceUsageType

The type of usage data to delete.
Type: String
Valid Values: `VOICE`
Required: Yes

Response Elements

If the action is successful, the service sends back an HTTP 200 response with an empty HTTP body.

Figure 6.2: Documentation for a method from the *Alexa for Business* API for the Alexa voice assistant. This method does not appear anywhere in code repositories on `github.com` as of the writing of this dissertation. To be able to use this method, models must learn to incorporate them into target source code based on their documentation alone.

users. An important and challenging area for future work is to generate source code explanations taking code context (or interaction history in the case of non-expert users) into account.

Learning from Background Knowledge The ubiquitous nature of software development today results in the release of several new API packages everyday, and it can take several months before their usage trickles into online code repositories and forums, on which our NL to code models rely to obtain training signals. However, most API packages are accompanied by exhaustive sets of documentation for every method and parameter. For example, Figure 6.2 shows documentation for the `delete_device_usage_data` method which was released as part of the recent *Alexa for Business* API for the Alexa voice assistant, and

d = 6 **NL:** Training a Decision Tree

d = 5 **NL:** Load features and labels in a dataframe.

d = 4
```
import pandas as pd
X = pd.read_json('features.json')
y = pd.read_json('labels.json')
```

d = 3 **NL:** Split the data into train and test.

d = 2
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

d = 1 **NL:** Create and train the model.

Target
```
from sklearn.tree import DecisionTreeClassifier
dtree = DecisionTreeClassifier()
dtree.fit(X_train, y_train)
```

Figure 6.3: This Python Jupyter notebook, comprising interleaved NL markdown and code cells, loads data from a file and trains a decision tree classifier. We aim to generate the target code cell (blue) based on the previous NL markdown *Create and train the model* and the prior NL and code history. To the left of each cell, *d* represents its distance from the target cell.

is used to clear any history of voice input from the device. This method does not appear anywhere in code repositories on `github.com` as of the writing of this dissertation. Models that aim to learn to successfully use these APIs must learn to incorporate them into target source code based on their documentation alone, without access to any online examples of their usage. This task is challenging because of the mismatch between the NL style used in code documentation (Figure 6.2) and the NL style used in NL instructions. This ability to apply APIs in a zero-shot sense i.e. having never seen their usage before, is vital for real world NL to source code systems, and is an important area for future research.

Interactive Programming Interactive programming with interleaved code snippet cells and natural language markdown is recently gaining popularity in the form of Jupyter notebooks, which accelerate prototyping and collaboration. Figure 6.3 shows part of a Jupyter notebook for training a decision tree classifier using Python, which includes a title, followed by interleaved NL markdown and code cells. Studying code

generation conditioned on a long context history in this setting is a potential new application of NL to code models, which introduces new challenges in addition to the ones discussed in this dissertation. We explore this setting in Agashe et al. [2019], where we present JuICe, a corpus of 1.5 million examples with a curated test set of 3.7K instances based on online programming assignments, together with baseline models for code cell generation, conditioned on the NL-Code history up to a particular code cell. We found that both increased context and distant supervision aid in code generation, but the dataset is still very challenging for current models. Improving NL to code models for this task is a highly impactful area for future work.

We hope that the steps we take in this dissertation towards efficient models and training paradigms for mapping NL to general purpose source code both for developers as well as non-expert users, together with all the future work it enables, serves to promote widespread research in this and related areas.

Bibliography

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5439–5449.

Andi Albrecht. 2015. python-sqlparse.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015a. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press.

Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM.

Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015b. Bimodal modelling of source code and natural language. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 2123–2132.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Jacob Andreas, Andreas Vlachos, and Stephen Clark. 2013. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 47–52.

Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81.

Gabor Angeli, Percy Liang, and Dan Klein. 2010. A simple domain-independent probabilistic approach to generation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 502–512.

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1699–1710. Association for Computational Linguistics.

Yoav Artzi and Luke Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Proceedings of the conference on empirical methods in natural language processing*, pages 421–432. Association for Computational Linguistics.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1(1):49–62.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 2015 International Conference on Learning Representations*, volume abs/1409.0473, San Diego, California. CBLS.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, volume 29, pages 65–72.

Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. Dbpal: A learned nl-interface for databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1765–1768. ACM.

J Berant, A Chou, F Roy, and L Percy. 2014. Freebase qa: Information extraction or semantic parsing. In *The 2014 Conference on Empirical Methods on Natural Language Processing*, pages 1511–1527.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544. Association for Computational Linguistics.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of ACL*, volume 7, page 92, Baltimore, Maryland. Association for Computational Linguistics.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM.

Qingqing Cai and Alexander Yates. 2013. Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 423–433.

Bo Chen, Le Sun, Xianpei Han, and Bo An. 2016a. Sentence rewriting for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 766–777, Berlin, Germany. Association for Computational Linguistics.

Danqi Chen, Jason Bolton, and Christopher D. Manning. 2016b. A thorough examination of the CNN/Daily Mail reading comprehension task. In *Association for Computational Linguistics (ACL)*.

David L Chen, Joohyun Kim, and Raymond J Mooney. 2010. Training a multilingual sportscaster: Using perceptual context to learn language. *Journal of Artificial Intelligence Research*, pages 397–435.

David L Chen and Raymond J Mooney. 2011. Learning to interpret natural language navigation instructions from observations. In *AAAI*, volume 2, pages 1–2.

Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 826–831. ACM.

Colin Cherry and George Foster. 2012. Batch tuning strategies for statistical machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 427–436.

Eunsol Choi, Tom Kwiatkowski, and Luke Zettlemoyer. 2015. Scalable semantic parsing with partial ontologies. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1311–1320. Association for Computational Linguistics.

James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. 2010. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, pages 18–27. Association for Computational Linguistics.

E. F. Codd. 1974. Seven steps to rendezvous with the casual user. In *IFIP Working Conference Data Base Management*, pages 179–200.

Fred J Damerau. 1985. Problems and some solutions in customization of natural language database front ends. *ACM Transactions on Information Systems (TOIS)*, 3(2):165–184.

Pradeep Dasigi. 2019. *Knowledge-Aware Natural Language Understanding*. Ph.D. thesis, Carnegie Mellon University Pittsburgh, PA.

Anne N De Roeck and Barry GT Lowden. 1986. Generating english paraphrases from formal relational calculus expressions. In *Proceedings of the 11th coference on Computational linguistics*, pages 581–583. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Jacob Devlin, Hao Cheng, Hao Fang, Saurabh Gupta, Li Deng, Xiaodong He, Geoffrey Zweig, and Margaret Mitchell. 2015. Language models for image captioning: The quirks and what works. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 100–105.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 731–742.

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431.

Catherine Finegan-Dollak, Jonathan K Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360.

Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. 2013. PPDB: The paraphrase database. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 758–764. Association for Computational Linguistics.

Alessandra Giordani and Alessandro Moschitti. 2012. Translating questions to SQL queries with generative parsers discriminatively reranked. In *Proceedings of COLING 2012: Posters*, pages 401–410. The COLING 2012 Organizing Committee.

Dan Goldwasser and Dan Roth. 2014. Learning from natural instructions. *Machine learning*, 94(2):205–232.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016a. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016b. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA. ACM.

Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM.

Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2019. Coupling retrieval and meta-learning for context-dependent semantic parsing. *arXiv preprint arXiv:1906.07108*.

Carolin Haas and Stefan Riezler. 2016. A corpus and semantic parser for multilingual natural language querying of openstreetmap. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 740–750.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source

code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930. Association for Computational Linguistics.

Yulan He and Steve Young. 2006. Spoken language understanding using the hidden vector state model. *Speech Communication*, 48(3-4):262–275.

Kenneth Heafield. 2011. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652. Association for Computational Linguistics.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22. Association for Computational Linguistics.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE Press.

Andrej Karpathy and Fei-Fei Li. 2015. Deep visual-semantic alignments for generating image descriptions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, pages 3128–3137.

Rohit J Kate and Raymond J Mooney. 2006. Using string-kernels for learning semantic parsers. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 913–920. Association for Computational Linguistics.

Chloé Kiddon, Luke Zettlemoyer, and Yejin Choi. 2016. Globally coherent text generation with neural checklist models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 329–339. Association for Computational Linguistics.

Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180.

Ioannis Konstas and Mirella Lapata. 2013. A global model for concept-to-text generation. *Journal of Artificial Intelligence Research*, 48(1):305–346.

Georgia Koutrika, Alkis Simitsis, and Yannis E Ioannidis. 2010. Explaining structured queries in natural language. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 333–344.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark. Association for Computational Linguistics.

Jayant Krishnamurthy and Tom Mitchell. 2012. Weakly supervised training of semantic parsers. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 754–765. Association for Computational Linguistics.

Nate Kushman, Yoav Artzi, Luke Zettlemoyer, and Regina Barzilay. 2014. Learning to automatically solve algebra word problems. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 271–281, Baltimore, Maryland. Association for Computational Linguistics.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836. Association for Computational Linguistics.

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1545–1556, Seattle, Washington, USA. Association for Computational Linguistics.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 1223–1233. Association for Computational Linguistics.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523. Association for Computational Linguistics.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, pages 795–806. IEEE Press.

Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. 2013. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1294–1303.

Fei Li and Hosagrahar V Jagadish. 2014. Nalir: An interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 709–712.

Percy Liang. 2016. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, 59(9):68–76.

Percy Liang, I. Michael Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2).

Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *Proceedings of the 20th international conference on Computational Linguistics*, page 501.

Kevin Lin, Ben Bogin, Mark Neumann, Jonathan Berant, and Matt Gardner. 2019. Grammar-based neural text-to-sql generation. *arXiv preprint arXiv:1905.13326*.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*.

Wang Ling, Phil Blunsom, Edward Grefenstette, Moritz Karl Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609. Association for Computational Linguistics.

Nicholas Locascio, Karthik Narasimhan, Eduardo De Leon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1918–1923, Austin, Texas. Association for Computational Linguistics.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856*.

Wei Lu and Hwee Tou Ng. 2011. A probabilistic forest-to-string model for language generation from typed lambda calculus expressions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1611–1622.

Thang Luong, Hieu Pham, and D. Christopher Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421. Association for Computational Linguistics.

Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating programming by example and natural language programming. In *AAAI*.

Hongyuan Mei, Mohit Bansal, and Matthew R. Walter. 2016. What to talk about and how? selective generation using lstms with coarse-to-fine alignment. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Scott Miller, David Stallard, Robert Bobrow, and Richard Schwartz. 1996. A fully statistical approach to natural language interfaces. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 55–61. Association for Computational Linguistics.

Kumar Dipendra Misra and Yoav Artzi. 2016. Neural shift-reduce CCG semantic parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1775–1786. Association for Computational Linguistics.

Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 35–40.

Eva-Maria M Mueckstein. 1983. Q-trans: Query translation into english. In *IJCAI*, pages 660–662. Citeseer.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*.

Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. 2013. Sorry, i don't speak sparql: Translating sparql queries into natural language. In *Proceedings of the 22Nd International Conference on World Wide Web*, pages 977–988.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. 2018. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318.

Kishore A Papineni, Salim Roukos, and Todd R Ward. 1997. Feature-based language understanding. In *Fifth European Conference on Speech Communication and Technology*.

Terence Parr. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

V. Pham, T. Bluche, C. Kermorvant, and J. Louradour. 2014. Dropout improves recurrent neural networks for handwriting recognition. In *2014 14th International Conference on Frontiers in Handwriting Recognition*, pages 285–290.

Hoifung Poon. 2013. Grounded unsupervised semantic parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 933–943. Association for Computational Linguistics.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157. ACM.

Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China. Association for Computational Linguistics.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems*, 3 edition. McGraw-Hill, Inc., New York, NY, USA.

Ganesh N Ramaswamy and Jan Kleindienst. 2000. Hierarchical feature-based translation for scalable natural language understanding. In *Sixth International Conference on Spoken Language Processing*.

Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. 2011. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 103–112.

Sarah Rastkar, Gail C Murphy, and Gabriel Murray. 2010. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 505–514.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association of Computational Linguistics*, 2(1):377–392.

Ehud Reiter and Robert Dale. 2000. *Building natural language generation systems*. Cambridge University Press, New York, NY.

Subhro Roy, Shyam Upadhyay, and Dan Roth. 2016. Equation parsing : Mapping sentences to grounded equations. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1088–1097. Association for Computational Linguistics.

Alexander M. Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 379–389.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41. ACM.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. *arXiv preprint arXiv:1906.10816*.

Alkis Simitsis and Yannis E. Ioannidis. 2009. Dbmss should talk back too. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Amanda Stent, Matthew Marge, and Mohit Singhai. 2005. Evaluating evaluation methods for generation in the presence of variation. In *Computational Linguistics and Intelligent Text Processing*, pages 341–351.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to map context-dependent sentences to executable formal queries. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, volume 1.

Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic parsing with syntax-and table-aware sql generation. *arXiv preprint arXiv:1804.08338*.

Ilya Sutskever, James Martens, and Geoffrey E Hinton. 2011. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024.

Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015*, pages 1556–1566.

Lappoon R Tang and Raymond J Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *European Conference on Machine Learning*, pages 466–477. Springer.

Bozena H Thompson and Frederick B Thompson. 1983. Introducing ask, a simple knowledgeable system. In *Proceedings of the first conference on Applied natural language processing*, pages 17–24. Association for Computational Linguistics.

Bozena H Thompson and Frederick B Thompson. 1985. Ask is transportable in half a dozen ways. *ACM Transactions on Information Systems (TOIS)*, 3(2):185–203.

Cynthia A. Thompson, Mary Elaine Califf, and Raymond J. Mooney. 1999. Active learning for natural language parsing and information extraction. In *Proceedings of the Sixteenth International Conference on Machine Learning*.

Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond J. Mooney, and Kate Saenko. 2015. Translating videos to natural language using deep recurrent neural networks. In *In Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1494–1504.

Adrienne Wang, Tom Kwiatkowski, and Luke Zettlemoyer. 2014. Morpho-syntactic lexical generalization for CCG semantic parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1284–1295. Association for Computational Linguistics.

I. Sida Wang, Percy Liang, and D. Christopher Manning. 2016. Learning language games through interaction. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2368–2378. Association for Computational Linguistics.

Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. 2017. Naturalizing a programming language via interactive learning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 929–938, Vancouver, Canada. Association for Computational Linguistics.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342. Association for Computational Linguistics.

Tsung-Hsien Wen, Milica Gasic, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. 2015. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1711–1721.

Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Wah Yuk Wong and Raymond Mooney. 2007. Generation by inverting a semantic parser that uses statistical machine translation. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 172–179. Association for Computational Linguistics.

Yuk Wah Wong and Raymond J Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 439–446. Association for Computational Linguistics.

William Woods. 1972. The lunar sciences natural language information system. *BBN report*.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, pages 2203–2214. ACM.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703. International World Wide Web Conferences Steering Committee.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206. Association for Computational Linguistics.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018a. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv preprint arXiv:1804.09769*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Wojciech Zaremba and Ilya Sutskever. 2014. Learning to execute. *CoRR*, abs/1410.4615.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*.

Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*.

Luke S Zettlemoyer and Michael Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 976–984. Association for Computational Linguistics.

Rui Zhang, Tao Yu, He Yang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Editing-based sql query generation for cross-domain context-dependent questions. *arXiv preprint arXiv:1909.00786*.

Xingxing Zhang and Mirella Lapata. 2014. Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 670–680.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.