Natural Language Processing Techniques for Code Generation

Hendrig Sellik
Delft University of Technology

ABSTRACT

Introduction: Software development is difficult and requires knowledge on many different levels such as understanding programming algorithms, languages, and frameworks. In addition, before code is being worked on, the system requirements and functionality are first discussed in natural language, after which it is sometimes visualized for the developers in a more formal language such as Unified Modeling Language.

Recently, researchers have tried to close the gap between natural language description of the system and the actual implementation in code using natural language processing techniques. The techniques from NLP have also proven to be useful at generating code snippets while developers work on source code. This literature survey aims to present an overview of the field of code generation using Natural Language Processing techniques.

Method: Google Scholar search engine was used to search for papers regarding code generation using NLP.

Results: A total of 428 abstracts were screened to reveal 36 papers suitable for the survey. The found papers were categorized into 6 groups by application type.

Conclusion: Source code has similarities to natural language, hence NLP techniques have been successfully used to generate code. Additionally, the area has also benefited from recent deep learning based advances in NLP.

KEYWORDS

Natural Language Processing, Code Generation, Software Engineering, Machine Learning, Deep Learning

1 INTRODUCTION

Software, particularly writing source code, requires a lot of technical background. Moreover, a person competent in one programming language is not necessarily good at others. One way to close the gap between the requirements written in natural language and the implemented software would be to generate the source code from the natural language, such as English. This would require less knowledge and enable writing software to non-developers such as personnel dealing with business requirements or help existing developers write code more effectively. In order to do so, researchers have been looking into *Natural Language Programming* [37] and using Natural Language Processing techniques to aid developers at different stages of software development.

While the area of Natural Language Processing has come a long way, generating code from Natural Language is still a major obstacle and the practice is not widely used in the industry. The aim of this paper is not only to gain knowledge about the subject but to also look into the effectiveness and current state of generating code from Natural Languages. This study includes papers from reputable peer-reviewed journals and conferences. To the best of the author's knowledge, no similar literature surveys have been attempted before and the results might help researchers to gain initial knowledge about the research conducted in the area of code generation using natural language processing techniques.

Structure In Section 2, the survey method is elaborated with search strategy and data extraction. In Section 3, the applications of the reviewed papers are described. In Section 4, some background knowledge is given about the main NLP techniques used in the reviewed papers. In Section 5, the sizes of the datasets are dicussed. Finally, the threats to validity are examined and the survey is concluded.

Research questions

The survey aims to answer the following research questions:

(1) What are the applications of the papers? In other words, what problems are being solved?
(2) What NLP techniques are used?
(3) How large are the datasets used in the selected papers?

2 SURVEY METHOD

This section describes the search criteria used to select the papers using the *Google Scholar* search engine and the followed data extraction process.

Search Criteria

Google Scholar search engine was used to search papers focusing on partial or full code generation based on natural language. The start date of the search was specified to be 2005 as the area of research is still maturing and knowledge about recent research was desired. In this literature study, the focus was on peer-reviewed conferences such as ICSE[1], FSE[2], ASE[3] and journals such as TSE[4], EMSE[5], TOSEM[6]. Hence

[1] http://www.icse-conferences.org/
[2] https://www.esec-fse.org
[3] https://2019.ase-conferences.org
[4] https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=32
[5] https://link.springer.com/journal/10664
[6] https://tosem.acm.org/

patents were not included in the search of Google Scholar. Only works in English were examined as it is the language that the author of this work is most familiar with and it is also the language used at the aforementioned journals.

All the criteria resulted in a following Google Scholar search query: *"natural language processing" OR "NLP" AND code generation source:ICSE OR source:FSE OR source:ASE OR source:TSE OR source:EMSE OR source:TOSEM*

The Google Scholar query returned a total of 100 results. The search strategy used is depicted in Figure 1. Titles and abstracts of the Google Scholar results were read to determine the most suitable papers for the literature survey. If the title and abstract did not contain enough information, full paper was read to determine suitability for data extraction.

Out of the 100 results returned from the original query, 13 results were suitable for the review. The backward snowballing [25] revealed another 328 papers (excluding duplicated papers) out of which 23 were selected after analyzing them as described above. This means that 36 papers in total were selected for the data extraction phase. A table of all 428 papers analyzed during the survey can be found in figshare[7].

Data Extraction

The following data was extracted from the chosen papers.

(1) Study reference
(2) Year of publication
(3) NLP technique used
(4) Dataset size (if applicable)
(5) Evaluation results
(6) Application domain

3 RQ 1: WHAT ARE THE APPLICATIONS OF THE PAPERS?

In this section, research applications using NLP techniques to generate code are discussed. In total, the applications were divided into 7 different categories:

(1) Generating identifier, method or class names
(2) Generating code comments
(3) Generating code snippets from natural language
(4) Searching for code using NLP techniques
(5) Generating pseudocode from source code
(6) Generating UML models
(7) Code completion

In Table 1, applications and the referenced papers are shown.

Generating Identifier, Method, and Class Names

Generating proper identifier, method, and class names consistently throughout the project is beneficial since it enhances the readability and maintainability of a project. Many researchers have tackled this issue.

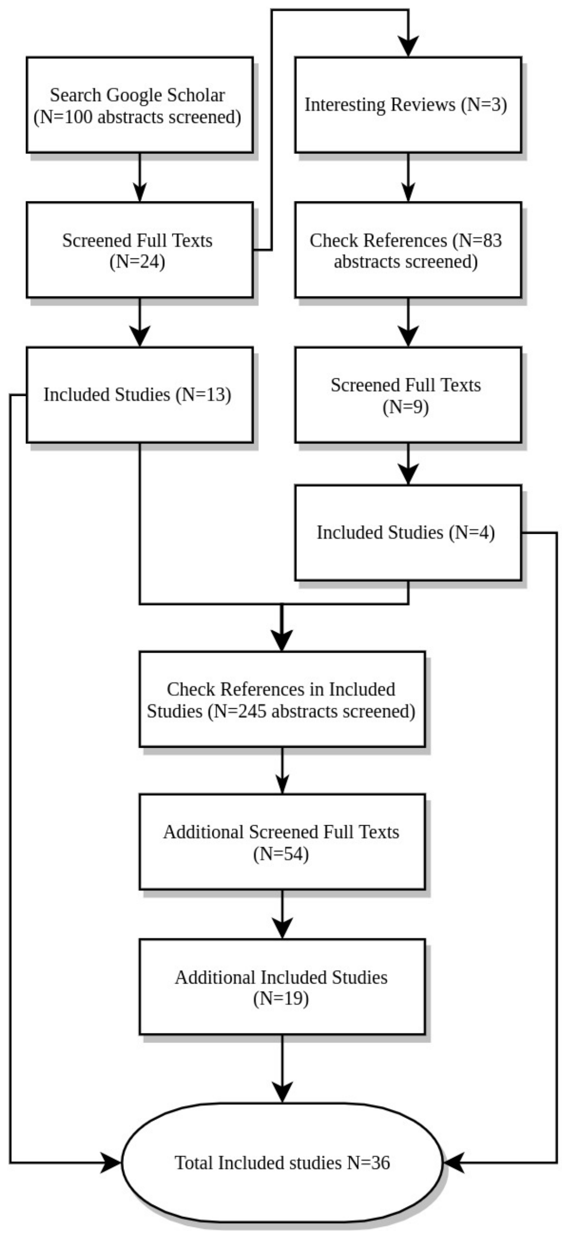

Figure 1: Flow chart of included studies. The numbers exclude duplicated papers.[47]

Allamanis et al. [2] and Lin et al. [34] created NATURALIZE for consistent identifier naming and LEAR for method naming respectively. They both use n-grams to generate tokens from source code, although LEAR does not use all textual tokens, but only lexical ones. They both later use different techniques to propose names based on the vocabulary. The authors of LEAR rated their results to be better than the ones by NATURALIZE, but the approach suffers from an unstable performance between different software projects.

[7] https://figshare.com/articles/Survey_NLP_for_Code_Generation_Screened_Papers_xls/11246510

Inspired by probabilistic language models used in NLP, Allamanis et al. [3] use a log-bilinear context model to generate names for methods. This means that they first use a model to assign each identifier in a method to a continuous vector space. Hence, similar identifiers are assigned with similar vectors or *embeddings* (see Section 4). After which the authors use the embeddings of the identifiers from the method body with the context model to predict method name word by word. This means that the model can use words not seen in the method body and combine these words in a way that has never been seen in the dataset. The model was trained on 20 popular GitHub projects and showed good results.

Alon et al. [5] further build on the idea of using word embeddings for method naming used by [3], however, they create an embedding for the entire method. Authors achieve it by parsing methods to Abstract Syntax Trees (AST-s). A deep learning model is then jointly trained to find the most important paths in the AST and also aggregate the paths into an embedding. This technique achieved a 75% improvement over previous techniques.

Generating Code Comments

Comments help to reduce software maintainability and therefore it is beneficial to have comments in source code.

Wong et al. [54] propose an automatic comment generator which extracts comments from Stack Overflow data dump and use tree-based NLP techniques to refine them. Finally, token-based clone detection is used to match pre-existing comments to code.

Sridhara et al. [49] generate comments to Java methods using abstract syntax trees and different NLP methods such as splitting, expanding abbreviations and extracting phrases from names. While it does not need a corpus like [54], the approach is very dependent on the proper method and variable naming in the source code. Movshovitz-Attias et al. [39] take on a simpler problem of comment-completion using n-grams and topic models, which proved to be more successful.

One of the earlier works to leverage the power of neural networks for code comment generation was by Iyer et al. [24]. They were inspired by the success of using neural networks for abstractive sentence summarization by Rush et al. [46]. Iyer at al. created CODE-NN, an end-to-end neural network that jointly performs content selection using an attention mechanism with Long Short Term Memory (LSTM) networks. The authors trained the network on C (66,015) and SQL (32,337) snippets from StackOverflow together with the title of the post. Although the model was simple and the dataset small, they managed to outperform the other state-of-the-art models at the time.

Hu et al. [23] create DeepCom which uses later advancements in NMT. They use a Seq2Seq model, also incorporating AST paths to generate comments. They use structure-based traversal to create a sequence of Java code AST to feed to DeepCom with the aim of better presenting the structure of the code compared to other traversal methods such as pre-order traversal. Work by Alon et al. [4] also takes a very similar approach to using a Seq2Seq model with a combination of AST. However, the key difference is the encoder. Instead of using a sequence of AST, the model creates a vector representation of each AST path which brings additional gains in precision.

Generating Code Snippets from Natural Language

While end-to-end source code generation from natural language would be ideal, no research articles achieving it were found. However, researchers are able to accomplish partial code generation.

Earlier work in the found literature use more simplistic statistical classification techniques. For example, Lei et al. [32] use the Bayesian generative model to generate C++ input parsers from natural language descriptions along with example input. Instead of generating code straight from the description, authors first convert it into a specification tree to map it into code. The tree is evaluated using their sampling framework which judges the correspondence of description with the tree in addition to the parsers' ability to parse the input.

Xiong et al. [55] use natural language processing to aid bug fix generation. The authors analyze Javadoc found for buggy methods and if possible, associate variable with an exception to produce a guard condition for it.

Mou et al. [38] use Recurrent Neural Networks (RNN) to generate small sections of code from natural language. However, the generated code is buggy, which makes the code hard to use.

An Eclipse IDE plugin by Nguyen et al. [40] is an effort to generate code from natural language. They extract data from Stack Overflow Data Dump [8], but instead of using information retrieval techniques, they leveraged a graph-based statistical machine translation model (SMT) to train on the corpus. The result was a novel tool that generated a graph of API elements that was synthesized into new source code. However, the resulting T2API tool is limited only to API-s while other solutions such as NLP2Code [9] can generate code from arbitrary natural language queries.

SWIM by Raghothaman et al. [44] is similar to T2API as it also uses SMT to train a model on clickthrough data from Bing and 25,000 open source project from GitHub. Although SWIM generates first code snippet in approximately 1.5 seconds and can be therefore considered to be usable, it lacks proper NLP techniques to distinguish similar API calls such as *Convert.ToDecimal* and *Convert.ToChar*. In addition, it is not language-agnostic by focusing only on C code and it was not implemented as an IDE plugin. Essentially, it is a less advanced version and predecessor of T2API.

Zhang et al. [57] generate test templates from test names. The test name and class of the code are given as input to

[8] https://archive.org/details/stackexchange

Table 1: Research Sorted by Application

Application	Reference
Generating Code Comments	[9], [49], [39], [24], [23]
Searching for Code Using NLP Techniques	[20], [9], [16]
Generating Pseudocode from Source Code	[15], [41]
Identifier, Method or Class naming	[2], [34], [3], [5], [4]
Code generation	[32], [55], [57], [18], [30], [43], [38], [40], [44]
Generating UML Models	[12], [26], [29], [17]
Code Completion	[8], [21], [45], [52], [14], [6], [33]

the system. The authors parse the name with the help of an external parsing system followed by identifying parts of the test relying on defined grammatical structure. Statistical analysis is used on a class to map parts of the test to the methods contained in the given class.

anyCode by Gvero et al. [18] is an Eclipse plugin that uses unigram and probabilistic context-free grammar model. The authors were able to synthesize small Java constructs and library invocations from a large GitHub Java corpus. The input is a mixture of natural language and Java code that a developer uses to query for code. While the input is flexible and anyCode can synthesize combinations of methods previously not seen in the corpus, the solution is limited by the examples provided to the model and it can not produce control flow constructs such as while loop and conditional statements.

The aforementioned code generation tools are aimed at developers and are at best realized as Eclipse plugins. However, there has been research which focuses on end-users as well. Le et al. [30] created an interactive interface to Smartphone users to create automatic scripts. It uses known NLP techniques such as bags-of-words, examining phrase length, punctuation and parse tree to map natural language text to API elements, after which program synthesis techniques are used to create an automation script similar to the automation tasks generated with Tasker [9]. Quirk et al. [43] use a log-linear model with character and word n-gram features to map natural language to if-this-then-that[10] code snippets using log-linear text classifier.

Searching for Code Using Natural Language Processing Techniques

While not strictly code generation, searching code with natural language enables developers to make simple queries against large code corpora, such as GitHub. Campbell et al. [9] also point out that developers spend a lot of time switching context between integrated developing environments (IDE) and web browsers to find suitable code snippets and complete a programming task.

Early work, such as the one by Hill et al. [20] use the concept of phrases instead of looking at each word separately. However, they heavily rely on the method signature, disregarding the information in the method body. In addition, the search query must be very close to the method signature.

Campbell et al. [9] propose an Eclipse plugin called *NLP2Code*, which enables developers to use their IDE to query for code using natural language statements such as *“add lines to a text file“*. The authors use 1,109,677 Stack Overflow threads tagged with "Java" from Stack Overflow data dump to find code snippets and their natural language descriptions. They also empirically tested the solution on undergraduate students and found the snippets to be helpful.

A very good example of the current capabilities is work by Gu et al. [16]. The authors use recent advancements in NLP such as word2vec (see Section 4) and neural machine translation to tie query and code semantics into the same vector space. This means that the queries can return semantically similar code instead of only returning similarly written code.

Generating Pseudocode from Source Code

When working with unfamiliar languages, it is useful to have a pseudo-code to better grasp the functionality of code.

Work by Fudaba et al. [15] and Oda et al. [41] convert Python code into an abstract syntax tree (AST), which is then translated to English or Japanese pseudocode using statistical machine translation.

[9] urlhttps://tasker.joaoapps.com
[10] https://ifttt.com/

Generating UML Models

Deeptimahanti et al. [12] use various NLP tools such as Stanford Parser[11], WordNet 2.1[12], Stanford Named Entity Recognizer[13] and JavaRAP [42] to create SUGAR - a tool which generates static UML models from natural language requirements. The authors prove the work by running SUGAR on a small text of stakeholder requirements which were consisted of simplistic sentences suitable for SUGAR. It was not tested on realistic project requirements that have more complicated, ambiguous or conflicting requirements.

As explained in Section 4, NLP methods have made significant progress. However, the survey did not find breakthrough progress in the area of generating UML models. Works such as [26], [29] from 2012 and a relevant paper by Gulia et al. [17] from 2016 still rely on POS tagging and WordNet. At the same time, they also rely on clear structures of specification text and manual rules to extract useful information.

Code Completion

A popular task among researchers is code completion since it is one of the most used features meant to aid developers. While code is not a natural language, research has been shown that code has properties inherent to natural language and therefore NLP techniques can be used to predict developer intent [21].

Bruch et al. [8] present the *best matching neighbor code completion system* (BMNCSS) to outperform the Eclipse code completion tool. The authors firstly capture the context of a variable by one-hot-encoding in a way where positive value is assigned to all methods and classes that call or encapsulate the variable. By comparing the distance of the vectors of code from existing corpora and the code to be completed, the authors were able to make suggestions based on the k-nearest neighbor algorithm. The algorithm is originally from pattern recognition but also used in NLP research [11].

Later, Hindle et al. take a different approach and compare source code to natural language [21]. They find that source code, like natural languages, is repetitive and predictable by statistical language models. They demonstrate the results by creating a simple n-gram model that outperforms Eclipse's code completion. While they do acknowledge that more advanced code completion algorithms existed at the time, like the previously mentioned BMN, they do not perform comparisons with their n-gram model. However, they share the vision of Bruch et al. [7] which states that the plethora of code available could be used for building models that help developers.

[11]The Stanford Natural Language Processing Group, Stanford Parser 1.6, http://nlp.stanford.edu/software/lex-parser.shtml

[12]Cognitive Science Laboratory, Princeton University, WordNet2.1, http://wordnet.princeton.edu/

[13]The Stanford Natural Language Processing Group, Stanford Named Entity Recognizer 1.0, http://nlp.stanford.edu/software/CRF-NER.shtml

Raychev et al. [45] reduce the problem of code completion to the natural language problem of predicting probabilities of sentences. The authors use models based on RNN and n-gram to fill holes regarding API usage and create a tool called *SLANG*. The best result is achieved by combining the two models with desired completion appearing in the top 3 results in 90% of the cases, proving that the approach is feasible for smaller tasks.

Tu et al. [52] find that while n-gram models can be successfully used on source code, they are unable to capture local regularities of source code that human-written projects have. Hence, they add a *cache-component* which works by having n-grams for local code and n-grams for the whole corpus. The final prediction is achieved by producing a linear interpolation of the two probabilities. Empirical testing verifies that this approach improves results.

Franks et al. [14] use the improved n-gram cache model and combine it with Eclipse's original code completion tool, creating *CACHEA*. The contribution combines the top 3 results of both of the models and improves the accuracy results of Tu et al. original cache model by 5% for suggestion results in the top 5.

Another solution to create a generative model for large scale and replace naive n-grams is proposed by Bielik et al. [6]. They navigate the Abstract Syntax Tree of code to create a probabilistic higher-order grammar (PHOG) which generalizes on probabilistic context-free grammar. This enables more precise results with the same training time. The authors consider it as a fundamental building block for other probabilistic models.

Li et al. [33] also make use of the AST-s, more specifically parent-child information. They use RNN with attention to deal with the long-range dependencies that previous similar models were having a hard time with. Because softmax neural language models have an output where each unique word corresponds to a dimension, these kinds of models use unknown tokens to limit vocabulary size. However, this is not useful for code completions.

To deal with the unknown word problem, a pointer mechanism was used. The authors note that usually, developers repeat the same tokens within the local code. The pointer mechanism uses this intuition and chooses a token from the local context to replace the unknown word. Both the attention and pointer mechanisms are techniques only recently adapted in NLP deep learning models (see Section 4).

4 RQ 2: WHAT NLP TECHNIQUES ARE USED?

In this section, different techniques used in NLP, such as different ML models and deep learning techniques are introduced as background. These are required to understand the key contributions of the analyzed papers. Many different NLP techniques were used and the most commonly used ones were selected for extension in this paper. Overall, the main NLP techniques divide into:

Table 2: Research Sorted by Technique

Method	Reference
Deep learning model	[38], [16],[33], [5], [3], [24], [23], [4], [45]
Embeddings	[3], [5], [23], [4], [16],[33], [24]
Statistical Machine Translation	[15], [41], [40], [44]
Machine Learning	[2] (SVM), [32] (Bayesian generative model)
n-gram model	[2], [39], [45], [52], [21], [14], [18], [43]
Probabilistic context-free grammar model	[18], [6],[49], [34]
One-hot-encoding relevant methods and classes surrounding the target code and using k-nearest neighbors algorithm to find similar vectors	[8]
Tree-based NLP techniques	[9], [30], [32]
Token-based clone detection	[9]
Word abbrevation	[49], [20], [29]
Word splitting	[49], [20], [17], [9], [57]
Word parsing	[49], [20], [17], [29], [26], [12], [55], [9], [57]
POS tagging	[17], [29], [29], [26], [12], [55], [9], [57]
WordNet	[17], [29], [26], [12]

(1) Widely used NLP Tools such as Part-of-Speech Taggers, Language Parsers and WordNet
(2) Statistical Machine Translation Models
(3) Word Embeddings
(4) Deep Learning Models

A summarizing overview of the papers using those techniques can be seen in Table 2.

Common NLP Tools

POS Tagging. Part-of-Speech (POS) Tagging [50] is used for assigning parts of speech, such as *noun*, *verb*, *adjective*, etc. to each word observed in the target text. There are several tools available for this, one commonly used tool is the Stanford Log-linear Part-Of-Speech Tagger[14].

Language Parser. Language Parsers are used to work out the grammatical structure of the sentence. For example, find which words in the sentence are subject and object to a noun or identify phrases in sentences [27]. A commonly used language parser for English is the Stanford Statistical Parser[15].

WordNet. WordNet [36] is a lexical database of English words where nouns, verbs, adjectives, and adverbs are grouped into sets of synonyms. Thus it allows identifying semantically similar words.

Statistical Machine Translation

Before Neural Machine Translation (NMT), the field of NLP was dominated by statistical machine translation. It is based on the idea that statistical models can be created from a bilingual text corpus. The models can be then used to create a most probable translation to a new text which the model has not seen before. The statistical models in NLP divide into word and phrase-based [56], syntax-based and structure-based [1].

At the beginning of the 2010s, neural network components were used in combination with the traditional statistical machine translation methods. However, Philipp Koehn emphasizes that when in 2015 there was only one pure neural machine translation model at the shared task for machine translation organized by the Conference on Machine Translation (WMT), then in 2016 neural systems won nearly all language pairs and in 2017 most of the submissions were neural systems [28].

Word Embeddings

Many earlier statistical and rule-based NLP systems regarded words as atomic symbols. This meant that simplistic models such as N-grams needed a lot of quality data. Mikolov et al. [35] made a novel contribution by assigning continuous vector presentations to words while preserving a relatively modest vector space (50-300). It was proved that one can get a lot of value by representing the meaning of a word by looking at the context in which it appears and taking advantage of that knowledge.

The Word2Vec model [35] features 2 different architectures, a Continuous Bag-of-Words Model to predict a word based on context. Secondly, the Skip-gram model is used to predict context based on a word.

[14]https://nlp.stanford.edu/software/tagger.shtml
[15]https://nlp.stanford.edu/software/lex-parser.shtml

Table 3: Research Sorted by Dataset Size

Dataset Size	Reference
No training data	[49], [20], [30], [26], [29], [17], [55], [57]
Not specified	[38], [40], [34]
106 problem descriptions totaling 424 sentences	[32]
18,805 Python and English pseudo-code pairs	[15], [41]
27,000 example usages	[8]
66,015 C# and 32,337 SQL code snippets with respective Stack-overflow title and posts	[24]
114K - 132K code-description pairs	[12], [43], [54]
2M tokens for n-gram model 5K tokens for local n-gram model	[52]
7-10 open-source Java Projects	[39], [21], [14], [3] (20 projects)
1,109,677 StackOverflow threads	[9]
JavaScript or Python code containing 150,000 files	[6], [33]
597.4MB equivalent to 6,989,349 snippets of code	[45]
12M- 18MJava methods	[5], [16], [4]
9,7K to 14.5K GitHub projects	[2], [23], [18], [44] (25K)

The main advantages of Word2Vec are that it is extremely fast compared to other similar solutions. In addition, the vector-word vocabulary gained from word2vec can be easily fed into neural networks or simply queried to detect similarity between words. Moreover, the solution is not only applicable to words, but to other text as well, such as source code. Hence its popularity in recent novel Software Engineering research solutions.

Deep Learning

The recent success of state-of-the-art NLP models is achieved by using deep learning models [13] and hence it is essential to understand them. A very good explanation of deep learning is provided by LeCun et al. [31] on which this section mostly relies on. The very basic Deep Learning solutions have an input vector fed into nodes of the first layer of deep neural networks. These are in turn connected to the next layer of nodes which are eventually connected to the output nodes. After data makes its way to output nodes, an objective function measures error (or distance) between the desired pattern and the actual pattern.

The main idea is that each node in the layers between input and output layer (also called hidden layers) has an adjustable parameter (or weight) which can be imagined as a knob that defines the input-output function. There could be millions of those, essentially turning deep learning systems into big functions. A model learns by calculating the objective function described earlier and trying to minimize the next output with a gradient vector to adjust weights in the hidden layers. Due to this structure, deep learning models also require a lot of data.

Although the individual values of the weights can be observed, they are a minuscule part of the whole and are therefore meaningless. Hence, some empirical testing is needed to find optimal parameters for models. More advanced models rearrange layers in different ways and add more features such as Long-Short-Term-Memory (LSTM)[22] and Gated Recurrent Units (GRU)[10] to increase effectiveness on longer sequences. Recently, breakthroughs have happened in the are of NLP thanks to attention, pointer and coverage mechanisms which all enhance the models' ability to deal with long-range dependencies [19, 48, 51, 53]. Additionally, models like BERT [13] allow training on a vast amount of corpora (3,3 billion words) and later use a small amount of computational resources and more a specialized corpus to fine-tune the model for a specific task.

5 RQ 3: HOW LARGE ARE THE DATASETS USED IN THE PAPERS?

It was found that different code generation techniques require datasets in various forms and sizes. This section is focused on the size of the datasets. The results of the findings can be seen in Table 3 which is approximately ordered by dataset size.

It can be seen that there are 8 papers that do not use training data at all. This means that the authors take advantage of grammatical rules and use the common NLP Tools discussed in Section 4. For some research, the dataset did exist, but the size could not be determined.

The rest of the datasets are quite diverse in size ranging from 106 problem-description pairs to 14,500 GitHub Java projects. It can be seen that while mining StackOverflow post-code pairs is quite popular, they usually range from 32,337 to 132,000. Only one paper uses 1,1 million Stackoverflow threads.

Papers using GitHub projects also have datasets diverse in size. While there are papers that use 7-20 GitHub projects, it can be seen that the datasets with the biggest size are also from GitHub containing 12M - 18M Java methods and the biggest datasets reach 9,700 - 14,500 GitHub projects. The latter ones are deep learning models with state-of-the-art results.

6 THREATS TO VALIDITY

Although the survey was constructed with the best systematic practices known to the author, there are some threats to validity.

Firstly, this survey was constructed in a limited time frame given by the course which means that there might be research that was not included in this survey. The final selection was confined to 36 papers.

Moreover, conferences such as ICML[16] and ICLR[17] were not included in the search criteria. These conferences may also contain research on software engineering. ICML especially may include state-of-the-art machine learning models with excellent results. This is an area of future research.

Finally, the author of the survey is not an expert in the field of code generation using NLP techniques. This affects the selection of the main techniques discussed in the survey, the categorization of the applications, also on selecting the papers to include in the survey and emphasizing research or techniques of some authors over the others. However, the author of this survey gave his best to give a thorough and complete overview.

7 CONCLUSION

Using techniques from the area of Natural Language Processing has proved to be successful at code generation and offers promising results. The area has followed the trends of NLP as the initial techniques such as POS tagging, n-gram and statistical models have been replaced by deep learning models in recent years. Code generation techniques from natural language have also greatly benefited from the recent advances in attention and pointer mechanisms which help with long-range dependencies. While using NLP techniques, researchers also take advantage of the structural nature of source code and use information from Abstract Syntax Trees.

While conducting the survey, the following observations were made which could help to advance the field:

(1) While the NLP domain has recently started to produce models that are trained on a vast amount of corpora and subsequently fine-tuned for a specific task, no similar research was found for source code. This could be an exciting future research area that has the potential to produce promising results.
(2) The area of UML model generation has seen no significant advancements since 2012 and the simplistic NLP techniques have remained almost the same. There is a potential to take advantage of the novel deep learning based techniques.
(3) The training and validations of the state-of-the-art code generation models were conducted on open-source software (OSS). No research was found focusing on industrial closed-source code. This is an important gap as a lot of solutions made for OSS might not work properly on closed-source code which has local characteristics.

As future research, this survey could be expanded to more conferences such as ICML or ICL and more abstracts could be scanned to get a more extensive overview of the field.

[16] https://icml.cc/

[17] https://iclr.cc/

REFERENCES

[1] Amr Ahmed and Greg Hanneman. 2005. Syntax-based statistical machine translation: A review. *Computational Linguistics* (2005).

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

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

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

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

[6] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. 2933–2942.

[7] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. 2010. IDE 2.0: collective intelligence in software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 53–58.

[8] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 213–222.

[9] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 628–632.

[10] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[11] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[12] Deva Kumar Deeptimahanti and Ratna Sanyal. 2008. An innovative approach for generating static UML models from natural language

requirements. In *International Conference on Advanced Software Engineering and Its Applications*. Springer, 147–163.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 705–708.

[15] Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Pseudogen: A tool to automatically generate pseudo-code from source code. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 824–829.

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[17] Sarita Gulia and Tanupriya Choudhury. 2016. An efficient automated design to generate UML diagram from Natural Language Specifications. In *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence)*. IEEE, 641–648.

[18] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Acm Sigplan Notices*, Vol. 50. ACM, 416–432.

[19] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *Advances in neural information processing systems*. 1693–1701.

[20] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 524–527.

[21] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

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

[23] 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*. ACM, 200–210.

[24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[25] Samireh Jalali and Claes Wohlin. 2012. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*. IEEE, 29–38.

[26] SD Joshi and Dhanraj Deshpande. 2012. Textual requirement analysis for UML diagram extraction by using NLP. *International journal of computer applications* 50, 8 (2012), 42–46.

[27] Dan Klein and Christopher D Manning. 2003. Fast exact inference with a factored model for natural language parsing. In *Advances in neural information processing systems*. 3–10.

[28] Philipp Koehn. 2017. Neural Machine Translation. *CoRR* abs/1709.07809 (2017). arXiv:1709.07809 http://arxiv.org/abs/1709.07809

[29] Poonam R Kothari. 2012. Processing natural language requirement to extract basic elements of a class. *International Journal of Applied Information Systems (IJAIS), ISSN* (2012), 2249–0868.

[30] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 193–206.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[32] 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)*. 1294–1303.

[33] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. AAAI Press, 4159–25. http://dl.acm.org/citation.cfm?id=3304222.3304348

[34] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, Gabriele Bavota, and Michele Lanza. 2017. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 81–90.

[35] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[36] George A Miller. 1998. *WordNet: An electronic lexical database.* MIT press.

[37] Lance A Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 20, 2 (1981), 184–215.

[38] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211* (2015).

[39] 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 (Volume 2: Short Papers)*, Vol. 2. 35–40.

[40] Thanh Nguyen, Peter C Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N Nguyen. 2016. T2API: synthesizing API code usage templates from English texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1013–1017.

[41] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[42] Long Qiu, Min-Yen Kan, and Tat-Seng Chua. 2004. A public reference implementation of the rap anaphora resolution algorithm. *arXiv preprint cs/0406031* (2004).

[43] 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)*. 878–888.

[44] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 357–367.

[45] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[46] Alexander M Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685* (2015).

[47] CE Scheepers, GCW Wendel-Vos, JM Den Broeder, EEMM Van Kempen, PJV Van Wesemael, and AJ Schuit. 2014. Shifting from car to active transport: a systematic review of the effectiveness of interventions. *Transportation research part A: policy and practice* 70 (2014), 264–280.

[48] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1073–1083. https://doi.org/10.18653/v1/P17-1099

[49] 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*. ACM, 43–52.

[50] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for computational Linguistics, 173–180.

[51] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. 2016. Modeling coverage for neural machine translation. *arXiv preprint arXiv:1601.04811* (2016).

[52] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 269–280.

[53] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*. 2692–2700.

[54] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.

[55] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[56] Richard Zens, Franz Josef Och, and Hermann Ney. 2002. Phrase-based statistical machine translation. In *Annual Conference on Artificial Intelligence*. Springer, 18–32.

[57] Benwen Zhang, Emily Hill, and James Clause. 2015. Automatically generating test templates from test names (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 506–511.

A RESEARCH ON NLP TECHNIQUES FOR CODE GENERATION

Table 4: Surveyed Research on NLP for Source Code Generation

Reference	Year	Method	Dataset Size	Evaluation Results	Application
Campbell et al. [54]	2013	Tree-based NLP to refine comments and token-based clone detection	132,767 code-description Mappings	-	Generating Code Comments
Sridhara et al. [49]	2010	AST and CFG for code, abbrevations, splitting, expanding for NL	No training	-	Generating Code Comments
Movshovitz et al. [39]	2013	Topic models and n-gram	9 open-source Java Projects	-	Code comment completion
Mou et al. [38]	2015	Seq2seq RNN model	Not specified	-	Generating Code Snippets from Natural language
Nguyen et al. [40]	2016	IBM Model (SMT) to map English texts and corresponding sequences of APIs. GraLan model to predict target code from NL	"large number" of StackOverflow posts	-	Generating Code Snippets from Natural language
Raghothaman et al. [44]	2016	Mine API usage patterns from GitHub projects and map them to queries from Bing based on clickthrough data	15 day click-through data From Bing and 25,000 GitHub projects	For 70% of the queries, The first suggested snippet Was a relevant solution (30 common queries)	Generating Code Snippets from Natural language
Hill et al. [20]	2011	Use method signatures from code to extract words and use the information to answer queries	No training	Precision 59%, Recall 38%, F-measure 37%	Searching for Code Using NLP Techniques
Campbell et al. [9]	2017	TaskNav algorithm and NLP techniques to map text query to code in StackOverflow posts	1,109,677 StackOverflow threads	74 queries out of 101 considered helpful by test subjects on the first invocation	Searching for Code Using NLP Techniques
Gu et al. [16]	2018	Create unified code and NL description embeddings using RNN, then use cosine similarity to find matches	18,233,872 Java methods from GitHub with English descriptions	0.46 success rate and precision on 1st try	Searching for Code Using NLP Techniques
Fudaba et al. [15]	2015	SMT techniques based on analyzing AST of source code	18,805 Python and English pseudo-code pairs	-	Generating Pseudocode from Source Code
Oda et al. [41]	2015	SMT techniques, especially phrase-based machine translation and tree-to-string machine translation from AST	18,805 Python and English pseudo-code pairs	54.08 BLEU score and 4.1/5 acceptability grade by developers	Generating Pseudocode from Source Code
Allamanis et al. [2]	2014	Train n-gram language model or SVM on code corpus (all code tokens), then rename identifiers in code to create candidates and list top suggestions based on model	10,968 open source projects	94% accuracy in top suggestions	Consistent Identifier Naming

Lin et al. [34]	2017	Static code analysis and NLP methods such as n-gram (on lexical code tokens)	Not specified	Precision 42.31% for suggested renaming	Consistent Method naming
Allamanis et al. [3]	2015	Log-bilinear neural network to model	20 GitHub projects	Outperforms n-gram models	Method and class naming
Alon et al. [5]	2019	Path-based neural attention model that simultaneously learns the embeddings of an AST paths and how to aggregate them to create an embedding of a method	12,000,000 Java methods	Precision 63.7, Recall 55.4, F1 Score 59.3	Method naming
Iyer et al. [24]	2016	An end-to-end neural network LSTM model with an attention mechanism	66,015 C# and 32,337 SQL code snippets with respective Stackoverflow title and posts	BLEU-4 20.5% for C# and 18.4% for SQL	Generating Code Comments
Hu et al. [23]	2018	Structure-based traversal is used to feed sequences of Java code AST paths to seq2seq Neural Machine Translation model	9,714 GitHub projects	BLEU-4 30% for C# and 30.94% for SQL	Generating Code Comments
Alon et al [4]	2018	Same as code2vec [5], but returns a sequence instead of classifying	16,000,000 Java methods	Precision 64.03, Recall 55.02, F1 59,19	Code summarization
Lei et al. [32]	2013	Bayesian generative model	106 problem descriptions totaling 424 sentences	Recall 72.5, Precision 89.3, F-Score 80.0	Generating C++ Input Parsers from NL description
Xiong et al. [55]	2017	Ranking of which one criterion is Javadoc analysis	No training	Precision 78.3, Recall 8.0	Generate guard conditions based on Javadoc
Zhang et al. [57]	2015	Analyzing the grammatical structure of a test name to create a test body	No training	Generating correct template from test name 80% of the time	Generating test templates from test names
Gvero et al. [18]	2015	Unigram and probabilistic context-free grammar model	14,500 Java projects containing over 1.8 million files from GitHub	Qualitative analysis concludes that the solution "often produces the expected code fragments"	Synthesizing Java code from NL and code mixture
Le et al. [30]	2013	Parse-trees, bag-of-words, regular expressions, phrase length measuring, punctuation detection	No training	Generating correct code for 90% of NL descriptions (640)	Create automation scripts for smartphones from NL description
Quirk et al. [43]	2015	A log-linear model with character and word n-gram features	114,408 code-description pairs	57.7% F1 score for producing correct snippet from NL description	Create if-this-then-that code snippets
Deeptimahanti et al. [12]	2008	Various NLP tools such as Stanford Parser, WordNet2.1, JavaRAP, etc. to identify actors, use cases and relationships to draw UML model	114,408 code-description pairs	Qualitatively analyzed on a simple description	Generating UML Models

Joshi et al. [26]	2012	Lexical and syntactic parser tools to analyze NL sentences, POS tagging, WordNet to identify actors and relationships. Java Application interface is used to extract domain ontology	No training	No evaluation	Generating UML Models
Kothari et al. [29]	2012	NLP tools for POS tagging, tokenization, WordNet2.1, identifying different actors and relations in UML by NLP	No training	No evaluation	Generating UML Models
Gulia et al. [17]	2016	Cleaning sentences, tokenization, detecting end-of-sentence, POS tagging and similar simplistic are used	No training	Qualitatively analyzed on a simple description	Generating UML Models
Bruch et al. [8]	2009	One-hot-encoding relevant methods and classes surrounding the target code and using k-nearest neighbors algorithm to find similar vectors	27,000 example usages	Recall 72%, Precision 82%, F1 Score 77%	Code Completion
Hindle et al. [21]	2012	N-gram language model	10 Apache Java projects	Useful completion found 67% more often if looked at the top 2 suggestions (compared to Eclipse default completion tool)	Code Completion
Raychev et al. [45]	2014	3-gram language model, RNNME-40, and the combination of the two	597.4MB equivalent to 6,989,349 snippets of code	90% of the cases the desired completion appears in the top 3 candidates	Code Completion
Tu et al. [52]	2014	Linear interpolation of global n-gram and local (cache) n-gram models	2M tokens for n-gram model 5K tokens for local n-gram model	65.07% MRR accuracy for suggestions in Apache Ant Java project	Code Completion
Franks et al. [14]	2015	Combine top 3 results from n-gram cache model and Eclipse suggestion	7 Apache projects between 60KLOC and 367KLOC	52.9% MRR accuracy for top1 suggestions in Apache Ant Java project (reported 48.11% for [52])	Code Completion
Bielik et al. [6]	2016	Generalized Probabilistic Context Free Grammar on AST-s	JavaScript code containing 150,000 files	18.5% error rate while predicting various JavaScript elements	Code Completion
Li et al. [33]	2018	LSTM based RNN with attention mechanism with added pointers to deal with unknown tokens. Heavily uses AST parent-child information	150,000 Javascript and Python program files	81% accuracy for the prediction of the next value for JavaScript code	Code Completion