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 discussed. 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 ICSE1, FSE2, ASE3 and journals such as TSE4, EMSE5, TOSEM6. Hence

1http://www.icse-conferences.org/
2https://www.esec-fse.org
3https://2019.ase-conferences.org
5https://link.springer.com/journal/10664
6https://tosem.acm.org/
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 figshare7.

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.

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.

Inspired by probabilistic language models used in NLP, Al-lamanis 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 (65,015) 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 4, 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

---

4https://archive.org/details/stackexchange
Table 1: Research Sorted by Application

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

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 Tasker⁹. Quirk et al. [43] use a log-linear model with character and word n-gram features to map natural language to if-this-then-that¹⁰ 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[https://tasker.joaoapps.com](https://tasker.joaoapps.com) ¹⁰[https://ifttt.com/](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.

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

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

(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\(^\text{14}\).

*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\(^\text{15}\).

*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.

\(^\text{14}\)https://nlp.stanford.edu/software/tagger.shtml
\(^\text{15}\)https://nlp.stanford.edu/software/lex-parser.shtml
Table 3: Research Sorted by Dataset Size

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

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 area 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 not be research that was not included in this survey. The final selection was confined to 36 papers.

Moreover, conferences such as ICML and ICLR 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.

REFERENCES


## Table 4: Surveyed Research on NLP for Source Code Generation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Year</th>
<th>Method</th>
<th>Dataset Size</th>
<th>Evaluation Results</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Campbell et al. [54]</td>
<td>2013</td>
<td>Tree-based NLP to refine comments and token-based clone detection</td>
<td>132,767 code-description Mappings</td>
<td>-</td>
<td>Generating Code Comments</td>
</tr>
<tr>
<td>Sridhara et al. [49]</td>
<td>2010</td>
<td>AST and CFG for code, abbreviations, splitting, expanding for NL</td>
<td>No training</td>
<td>-</td>
<td>Generating Code Comments</td>
</tr>
<tr>
<td>Movshovitz et al. [39]</td>
<td>2013</td>
<td>Topic models and n-gram</td>
<td>9 open-source Java Projects</td>
<td>-</td>
<td>Code comment completion</td>
</tr>
<tr>
<td>Mou et al. [38]</td>
<td>2015</td>
<td>Seq2seq RNN model</td>
<td>Not specified</td>
<td>-</td>
<td>Generating Code Snippets from Natural language</td>
</tr>
<tr>
<td>Nguyen et al. [40]</td>
<td>2016</td>
<td>IBM Model (SMT) to map English texts and corresponding sequences of APIs. GraLan model to predict target code from NL</td>
<td>&quot;large number&quot; of StackOverflow posts</td>
<td>-</td>
<td>Generating Code Snippets from Natural language</td>
</tr>
<tr>
<td>Raghothaman et al. [44]</td>
<td>2016</td>
<td>Mine API usage patterns from GitHub projects and map them to queries from Bing based on clickthrough data</td>
<td>15 day click-through data From Bing and 25,000 GitHub projects</td>
<td>For 70% of the queries, The first suggested snippet Was a relevant solution (30 common queries)</td>
<td>Generating Code Snippets from Natural language</td>
</tr>
<tr>
<td>Hill et al. [20]</td>
<td>2011</td>
<td>Use method signatures from code to extract words and use the information to answer queries</td>
<td>No training</td>
<td>Precision 59%, Recall 38%, F-measure 37%</td>
<td>Searching for Code Using NLP Techniques</td>
</tr>
<tr>
<td>Campbell et al. [9]</td>
<td>2017</td>
<td>TaskNav algorithm and NLP techniques to map text query to code in StackOverflow posts</td>
<td>1,109,677 StackOverflow threads</td>
<td>74 queries out of 101 considered helpful by test subjects on the first invocation</td>
<td>Searching for Code Using NLP Techniques</td>
</tr>
<tr>
<td>Gu et al. [16]</td>
<td>2018</td>
<td>Create unified code and NL description embeddings using RNN, then use cosine similarity to find matches</td>
<td>18,233,872 Java methods from GitHub with English descriptions</td>
<td>0.46 success rate and precision on 1st try</td>
<td>Searching for Code Using NLP Techniques</td>
</tr>
<tr>
<td>Oda et al. [41]</td>
<td>2015</td>
<td>SMT techniques, especially phrase-based machine translation and tree-to-string machine translation from AST</td>
<td>18,805 Python and English pseudo-code pairs</td>
<td>54.08 BLEU score and 4.1/5 acceptability grade by developers</td>
<td>Generating Pseudocode from Source Code</td>
</tr>
<tr>
<td>Allamanis et al. [2]</td>
<td>2014</td>
<td>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</td>
<td>10,968 open source projects</td>
<td>94% accuracy in top suggestions</td>
<td>Consistent Identifier Naming</td>
</tr>
<tr>
<td>Authors</td>
<td>Year</td>
<td>Description</td>
<td>Dataset Details</td>
<td>Results</td>
<td>Use Case</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Lin et al. [34]</td>
<td>2017</td>
<td>Static code analysis and NLP methods such as n-gram (on lexical code tokens)</td>
<td>Not specified</td>
<td>Precision 42.31% for suggested renaming</td>
<td>Consistent Method naming</td>
</tr>
<tr>
<td>Allamanis et al. [3]</td>
<td>2015</td>
<td>Log-bilinear neural network to model</td>
<td>20 GitHub projects</td>
<td>Outperforms n-gram models</td>
<td>Method and class naming</td>
</tr>
<tr>
<td>Alon et al. [5]</td>
<td>2019</td>
<td>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</td>
<td>12,000,000 Java methods</td>
<td>Precision 63.7, Recall 55.4, F1 Score 59.3</td>
<td>Method naming</td>
</tr>
<tr>
<td>Iyer et al. [24]</td>
<td>2016</td>
<td>An end-to-end neural network LSTM model with an attention mechanism</td>
<td>66,015 C# and 32,337 SQL code snippets with respective Stackoverflow title and posts</td>
<td>BLEU-4 20.5% for C# and 18.4% for SQL</td>
<td>Generating Code Comments</td>
</tr>
<tr>
<td>Hu et al. [23]</td>
<td>2018</td>
<td>Structure-based traversal is used to feed sequences of Java code AST paths to seq2seq Neural Machine Translation model</td>
<td>9,714 GitHub projects</td>
<td>BLEU-4 30% for C# and 30.94% for SQL</td>
<td>Generating Code Comments</td>
</tr>
<tr>
<td>Alon [4]</td>
<td>2018</td>
<td>Same as code2vec [5], but returns a sequence instead of classifying</td>
<td>16,000,000 Java methods</td>
<td>Precision 64.03, Recall 55.02, F1 59.19</td>
<td>Code summarization</td>
</tr>
<tr>
<td>Lei et al. [32]</td>
<td>2013</td>
<td>Bayesian generative model</td>
<td>106 problem descriptions totaling 424 sentences</td>
<td>Recall 72.5, Precision 89.3, F-Score 80.0</td>
<td>Generating C++ Input Parsers from NL description</td>
</tr>
<tr>
<td>Xiong et al. [55]</td>
<td>2017</td>
<td>Ranking of which one criterion is Javadoc analysis</td>
<td>No training</td>
<td>Precision 78.3, Recall 8.0</td>
<td>Generate guard conditions based on Javadoc</td>
</tr>
<tr>
<td>Zhang et al. [57]</td>
<td>2015</td>
<td>Analyzing the grammatical structure of a test name to create a test body</td>
<td>No training</td>
<td>Generating correct template from test name 80% of the time</td>
<td>Generating test templates from test names</td>
</tr>
<tr>
<td>Gvero et al. [18]</td>
<td>2015</td>
<td>Unigram and probabilistic context-free grammar model</td>
<td>14,500 Java projects containing over 1.8 million files from GitHub</td>
<td>Qualitative analysis concludes that the solution &quot;often produces the expected code fragments&quot;</td>
<td>Synthesizing Java code from NL and code mixture</td>
</tr>
<tr>
<td>Le et al. [30]</td>
<td>2013</td>
<td>Parse-trees, bag-of-words, regular expressions, phrase length measuring, punctuation detection</td>
<td>No training</td>
<td>Generating correct code for 90% of NL descriptions (640)</td>
<td>Create automation scripts scripts for smartphones from NL description</td>
</tr>
<tr>
<td>Quirk et al. [43]</td>
<td>2015</td>
<td>A log-linear model with character and word n-gram features</td>
<td>114,408 code-description pairs</td>
<td>57.7% F1 score for producing correct snippet from NL description</td>
<td>Create if-this-then-that code snippets</td>
</tr>
<tr>
<td>Deeptimahanti et al. [12]</td>
<td>2008</td>
<td>Various NLP tools such as Stanford Parser, WordNet2.1, JavaRAP, etc. to identify actors, use cases and relationships to draw UML model</td>
<td>114,408 code-description pairs</td>
<td>Qualitatively analyzed on a simple description</td>
<td>Generating UML Models</td>
</tr>
</tbody>
</table>

12
<table>
<thead>
<tr>
<th>Authors</th>
<th>Year</th>
<th>Methodology</th>
<th>Learning</th>
<th>Evaluation</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joshi et al. [26]</td>
<td>2012</td>
<td>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</td>
<td>No training</td>
<td>No evaluation</td>
<td>Generating UML Models</td>
</tr>
<tr>
<td>Kothari et al. [29]</td>
<td>2012</td>
<td>NLP tools for POS tagging, tokenization, WordNet2.1, identifying different actors and relations in UML by NLP</td>
<td>No training</td>
<td>No evaluation</td>
<td>Generating UML Models</td>
</tr>
<tr>
<td>Gulia et al. [17]</td>
<td>2016</td>
<td>Cleaning sentences, tokenization, detecting end-of-sentence, POS tagging and similar simplistic are used</td>
<td>No training</td>
<td>Qualitatively analyzed on a simple description</td>
<td>Generating UML Models</td>
</tr>
<tr>
<td>Bruch et al. [8]</td>
<td>2009</td>
<td>One-hot-encoding relevant methods and classes surrounding the target code and using k-nearest neighbors algorithm to find similar vectors</td>
<td>27,000 example usages</td>
<td>Recall 72%, Precision 82%, F1 Score 77%</td>
<td>Code Completion</td>
</tr>
<tr>
<td>Hindle et al. [21]</td>
<td>2012</td>
<td>N-gram language model</td>
<td>10 Apache Java projects</td>
<td>Useful completion found 67% more often if looked at the top 2 suggestions (compared to Eclipse default completion tool)</td>
<td>Code Completion</td>
</tr>
<tr>
<td>Raychev et al. [45]</td>
<td>2014</td>
<td>3-gram language model, RNNME-40, and the combination of the two</td>
<td>597.4MB equivalent to 6,989,349 snippets of code</td>
<td>90% of the cases the desired completion appears in the top 3 candidates</td>
<td>Code Completion</td>
</tr>
<tr>
<td>Tu et al. [52]</td>
<td>2014</td>
<td>Linear interpolation of global n-gram and local (cache) n-gram models</td>
<td>2M tokens for n-gram model 5K tokens for local n-gram model</td>
<td>65.07% MRR accuracy for suggestions in Apache Ant Java project</td>
<td>Code Completion</td>
</tr>
<tr>
<td>Franks et al. [14]</td>
<td>2015</td>
<td>Combine top 3 results from n-gram cache model and Eclipse suggestion</td>
<td>7 Apache projects between 60KLOC and 367KLOC</td>
<td>52.9% MRR accuracy for top1 suggestions in Apache Ant Java project (reported 48.11% for [52])</td>
<td>Code Completion</td>
</tr>
<tr>
<td>Bielik et al. [6]</td>
<td>2016</td>
<td>Generalized Probabilistic Context Free Grammar on AST-s</td>
<td>Javascript code containing 150,000 files</td>
<td>18.5% error rate while predicting various JavaScript elements</td>
<td>Code Completion</td>
</tr>
<tr>
<td>Li et al. [33]</td>
<td>2018</td>
<td>LSTM based RNN with attention mechanism with added pointers to deal with unknown tokens. Heavily uses AST parent-child information</td>
<td>150,000 Javascript and Python program files</td>
<td>81% accuracy for the prediction of the next value for JavaScript code</td>
<td>Code Completion</td>
</tr>
</tbody>
</table>