A Survey of Machine Learning for Big Code and Naturalness

MILTIADIS ALLAMANIS, Microsoft Research
EARL T. BARR, University College London
PREMKUMAR DEVANBU, University of California, Davis
CHARLES SUTTON, University of Edinburgh and The Alan Turing Institute

Research at the intersection of machine learning, programming languages, and software engineering has recently taken important steps in proposing learnable probabilistic models of source code that exploit code's abundance of patterns. In this article, we survey this work. We contrast programming languages against natural languages and discuss how these similarities and differences drive the design of probabilistic models. We present a taxonomy based on the underlying design principles of each model and use it to navigate the literature. Then, we review how researchers have adapted these models to application areas and discuss cross-cutting and application-specific challenges and opportunities.

CCS Concepts: • **Computing methodologies** → **Machine learning**; Natural language processing; • **Software and its engineering** → **Software notations and tools**; • **General and reference** → *Surveys and overviews*;

Additional Key Words and Phrases: Big Code, Code Naturalness, Software Engineering Tools, Machine Learning

1 INTRODUCTION

Software is ubiquitous in modern society. Almost every aspect of life, including healthcare, energy, transportation, public safety, and even entertainment, depends on the reliable operation of high-quality software. Unfortunately, developing software is a costly process: software engineers need to tackle the inherent complexity of software while avoiding bugs, and still delivering highly functional software products on time. There is therefore an ongoing demand for innovations in software tools that help make software more reliable and maintainable. New methods are constantly sought, to reduce the complexity of software and help engineers construct better software.

Research in this area has been dominated by the *formal*, or *logico-deductive*, approach. Practitioners of this approach hold that, since software is constructed in mathematically well-defined programming languages, software tools can be conceived in purely formal terms. The design of software tools is to be approached using formal methods of definition, abstraction, and deduction. Properties of tools thus built should be proven using rigorous proof techniques such as induction over discrete structures. This logico-deductive approach has tremendous appeal in programming languages research, as it holds the promise of proving facts and properties of the program. Many elegant and powerful abstractions, definitions, algorithms, and proof techniques have been developed, which have led to important practical tools for program verification, bug finding, and refactoring [24, 42, 45]. It should be emphasized that these are theory-first approaches. Software constructions are viewed primarily as mathematical objects, and when evaluating software tools

This work was supported by Microsoft Research Cambridge through its PhD Scholarship Programme. M. Allamanis, E. T. Barr, and C. Sutton are supported by the Engineering and Physical Sciences Research Council [grant numbers EP/K024043/1, EP/P005659/1, and EP/P005314/1]. P. Devanbu is supported by the National Research Foundation award number 1414172.
Author's addresses: M. Allamanis, Microsoft Research, Cambridge, UK; E.T. Barr, University College London, UK; P. Devanbu, UC Davis, CA, USA; C. Sutton, University of Edinburgh, UK and Alan Turing Institute, London, UK; .

built using this approach, the elegance and rigor of definitions, abstractions, and formal proofs-of-properties are of dominant concern. The actual varieties of *use* of software constructs, in practice, become relevant later, in case studies, that typically accompany presentations in this line of work.

Of late, another valuable resource has arisen: the large and growing body of successful, widely used, open-source software systems. Open-source software systems such as Linux, MySQL, Django, Ant, and OpenEJB have become ubiquitous. These systems publicly expose not just source code, but also meta-data concerning authorship, bug-fixes, and review processes. The scale of available data is massive: billions of tokens of code and millions of instances of meta-data, such as changes, bug-fixes, and code reviews ("big code"). The availability of "big code" suggests a new, data-driven approach to developing software tools: why not let the statistical distributional properties, estimated over large and representative software corpora, also influence the design of development tools? Thus rather than performing well in the worst case, or in case studies, our tools can perform well in *most cases*, thus delivering greater advantages in expectation. The appeal of this approach echoes that of earlier work in computer architecture: Amdahl's law [15], for example, tells us to focus on the common case. This motivates a similar hope for development tools, that tools for software development and program analysis can be improved by focusing on the common cases using a fine-grained estimate of the statistical distribution of code. Essentially, the hope is that analyzing the text of thousands of well-written software projects can uncover patterns that partially characterize software that is reliable, easy to read, and easy to maintain.

The promise and power of machine learning rests on its ability to generalize from examples and handle noise. To date, software engineering (SE) and programming languages (PL) research has largely focused on using machine learning (ML) techniques as black boxes to replace heuristics and find features, sometimes without appreciating the subtleties of the assumptions these techniques make. A key contribution of this survey is to elucidate these assumptions and their consequences. Just as natural language processing (NLP) research changed focus from brittle rule-based expert systems that could not handle the diversity of real-life data to statistical methods [99], SE/PL should make the same transition, augmenting traditional methods that consider only the formal structure of programs with information about the statistical properties of code.

Structure. First, in Section 2, we discuss the basis of this area, which we call the "naturalness hypothesis". We then review recent work on machine learning methods for analyzing source code, focusing on probabilistic models, such as n-gram language models and deep learning methods.[1] We also touch on other types of machine learning-based source code models, aiming to give a broad overview of the area, to explain the core methods and techniques, and to discuss applications in programming languages and software engineering. We focus on work that goes beyond a "bag of words" representation of code, modeling code using sequences, trees, and continuous representations. We describe a wide range of emerging applications, ranging from recommender systems, debugging, program analysis, and program synthesis. The large body of work on semantic parsing [138], is not the focus of this survey but we include some methods that output code in general-purpose programming languages (rather than carefully crafted domain-specific languages). This review is structured as follows. We first discuss the different characteristics of natural language and source code to motivate the design decisions involved in machine learning models of code (Section 3). We then introduce a taxonomy of probabilistic models of source code (Section 4). Then we describe the software engineering and programming language applications of probabilistic

[1]It may be worth pointing out that deep learning and probabilistic modeling are *not* mutually exclusive. Indeed, many of the currently most effective methods for language modeling, for example, are based on deep learning.

source code models (Section 5). Finally, we mention a few overlapping research areas (Section 7), and we discuss challenges and interesting future directions (Section 6).

Related Reviews and other Resources. There have been short reviews summarizing the progress and the vision of the research area, from both software engineering [52] and programming languages perspectives [28, 195]. However, none of these articles can be considered extensive literature reviews, which is the purpose of this work. Ernst [57] discusses promising areas of applying natural language processing to software development, including error messages, variable names, code comments, and user questions. Some resources, datasets and code can be found at http://learnbigcode.github.io/. An online version of the work reviewed here — which we will keep up-to-date by accepting external contributions — can be found at https://ml4code.github.io.

2 THE NATURALNESS HYPOTHESIS

Many aspects of code, such as names, formatting, the lexical order of methods, *etc.* have no impact on program semantics. This is precisely why we abstract them in most program analyses. But then, why should statistical properties of code matter at all? To explain this, we recently suggested a hypothesis, called the *naturalness hypothesis*. The inspiration for the naturalness hypothesis can be traced back to the "literate programming" concept of D. Knuth, which draws from the insight that programming is a form of human communication: *"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do..." [105]* The naturalness hypothesis, then, holds that

> **The naturalness hypothesis.** *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.*

The exploitation of the statistics of human communication is a mature and effective technology, with numerous applications [99]. Large corpora of human communication, *viz.* natural language corpora, have been extensively studied, and highly refined statistical models of these corpora have been used to great effect in speech recognition, translation, error-correction, *etc.*.

The naturalness hypothesis holds that, because coding is an act of communication, one might expect large code corpora to have rich patterns, similar to natural language, thus allowing software engineering tools to exploit probabilistic ML models. The first empirical evidence of this hypothesis, showing that models originally developed for natural language were surprisingly effective for source code, was presented by Hindle et al. [87, 88]. More evidence and numerous applications of this approach have followed, which are the subject of this review.

The naturalness hypothesis, then, inspires the goal to apply machine learning approaches to create probabilistic source code models that *learn* how developers *naturally* write and use code. These models can be used to augment existing tools with statistical information and enable new machine learning-based software engineering tools, such as recommender systems and program analyses. At a high level, statistical methods allow a system to make hypotheses, along with probabilistic confidence values, of what a developer *might* want to do next or what formal properties *might* be true of a chunk of code. Probabilistic methods also provide natural ways of learning correspondences between code and other types of documents, such as requirements, blog posts, comments, *etc.*— such correspondences will always be uncertain, because natural language is ambiguous, and so the quantitative measure of confidence provided by probabilities is especially natural. As we discuss in Section 5, one could go so far as to claim that almost every area of software engineering and programming language research has potential opportunities for exploiting statistical properties.

Although the "naturalness hypothesis" may not seem surprising, one should appreciate the root cause of "naturalness". Naturalness of code seems to have a strong connection with the fact that developers prefer to write [5] and read [85] code that is conventional, idiomatic, and familiar because it helps understanding and maintaining software systems. Code that takes familiar forms is more *transparent*, in that its meaning is more readily apparent to an experienced reader. Thus, the naturalness hypothesis leads seamlessly to a "code predictability" notion, suggesting that code artifacts — from simple token sequences to formal verification statements — contain useful recurring and predictable patterns that can be exploited. "Naturalness" and "big code" should be viewed as instances of a more general concept that there is exploitable regularity across human-written code that can be "absorbed" and generalized by a learning component that can transfer its knowledge and probabilistically reason about new code.

This article reviews the emerging area of machine learning and statistical natural language processing methods applied to source code. We focus on probabilistic models of code, that is, methods that estimate a distribution over all possible source files. Machine learning in probabilistic models has seen wide application throughout artificial intelligence, including natural language processing, robotics, and computer vision, because of its ability to handle uncertainty and to learn in the face of noisy data. One might reasonably ask why it is necessary to handle uncertainty and noise in software development tools, when in many cases the program to be analyzed is known (there is no uncertainty about what the programmer has written) and is deterministic. In fact, there are several interesting motivations for incorporating probabilistic modeling into machine learning methods for software development. First, probabilistic methods offer a principled method for handling uncertainty and fusing multiple, possibly ambiguous, sources of information. Second, probabilistic models provide a natural framework for connecting prior knowledge to data — providing a natural framework to design methods based on abstractions of statistical properties of code corpora. In particular, we often wish to infer relationships between source code and natural language text, such as comments, bug reports, requirements documents, documentation, search queries, and so on. Because natural language text is ambiguous, it is useful to quantify uncertainty in the correspondence between code and text. Finally, when predicting program properties, probabilities provide a way to relax strict requirements on soundness: we can seek unsound methods that predict program properties based on statistical patterns in the code, using probabilities as a way to quantify the method's confidence in its predictions.

3 TEXT, CODE AND MACHINE LEARNING

Programming languages narrow the gap between computers and the human mind: they construct palatable abstractions out of a multitude of minute state transitions. Source code has two audiences and is inherently *bimodal*: it communicates along two channels: one with humans, and one with computers. Humans must understand code to read and write it; computers must be able to execute it. The bimodality of code drives the similarities and differences between it and text. Below, we discuss these similarities and differences with forward pointers to how they have been exploited, handled, or remain open. Although code and text are similar, code written in a general-purpose programming languages, is a relatively new problem domain for existing ML and NLP techniques. Hindle et al. [87] not only showed that exploitable similarity exists between the two via an n-gram language model, but that code is even less surprising than text. Although it may seem manifestly obvious that code and text have many differences, it is useful to enumerate these differences carefully, as this allows us to gain insight into when techniques from NLP need to be modified to deal with code. Perhaps the most obvious difference is that code is executable and has formal syntax and semantics. We

close by discussing how source code bimodality manifests itself as synchronization points between the algorithmic and explanatory channels.

Executability. All code is executable; text often is not. So code is often semantically brittle — small changes (*e.g.* swapping function arguments) can drastically change the meaning of code; whereas natural language is more robust in that readers can often understand text even if it contains mistakes. Despite the bimodal nature of code and its human-oriented modality, the sensitivity of code semantics to "noise" necessitates the combination of probabilistic and formal methods. For example, existing work builds probabilistic models then applies strict formal constraints to filter their output (Section 4.1) or uses them to guide formal methods (Section 5.8). Nevertheless, further research on bridging formal and probabilistic methods is needed (Section 6.1).

Whether it is possible to translate between natural languages in a way that completely preserves meaning is a matter of debate. Programming languages, on the other hand, can be translated between each other exactly, as all mainstream programming languages are Turing-complete. (That said, porting real-world programs to new languages and platforms remains challenging in practice [32]). ML techniques have not yet comprehensively tackled such problems and are currently limited to solely translating among languages with very similar characteristics, *e.g.* Java and C# (Section 6.1). Programming languages differ in their expressivity and intelligibility, ranging from Haskell to Malbolge[2], with some especially tailored for certain problem domains; in contrast, natural languages are typically used to communicate across a wide variety of domains. Executability of code induces control and data flows within programs, which have only weak analogs in text. Finally, executability gives rise to additional modalities of code — its static and dynamic views (*e.g.* execution traces), which are not present in text. Learning over traces or flows are promising directions (Section 6.1).

Formality. Programming languages are formal languages, whereas formal languages are only mathematical models of natural language. As a consequence, programming languages are designed top-down by a few designers for many users. Natural languages, in contrast, emerge, bottom up, "through social dynamics" [46]. Natural languages change gradually, while programming languages exhibit punctuated change: new releases, like Python 3, sometimes break backward compatibility. Formatting can also be meaningful in code: Python's whitespace sensitivity is the canonical example. Text has a robust environmental dependence, whereas code suffers from bit rot — the deterioration of software's functionality through time because of changes in its environment (*e.g.* dependencies) — because all its explicit environmental interactions must be specified upfront and execution environments evolve much more quickly than natural languages.

Source code's formality facilities reuse. Solving a problem algorithmically is cognitively expensive, so developers actively try to reuse code [95], moving common functionality into libraries to facilitate reuse. As a result, usually functions are semantically unique within a project. Coding competitions or undergraduate projects are obvious exceptions. In contrast, one can find thousands of news articles describing an important global event. On the other hand, Gabel and Su [67] have found that locally, code is more pattern dense than text (Section 4.1). This has led to important performance improvements on some applications, such as code completion (Section 5.1).

Because programming languages are automatically translated into machine code, they must be syntactically, even to a first approximation semantically, unambiguous[3]. In contrast to NLP models, which must always account for textual ambiguity, probabilistic models of code can and do take advantage of the rich and unambiguous code structure. Although it is less pervasive, ambiguity

[2]https://en.wikipedia.org/wiki/Malbolge

[3]Exceptions exist, like http://po-ru.com/diary/ruby-parsing-ambiguities/ and https://stackoverflow.com/questions/38449606/ambiguity-in-multiple-inheritance, but these rarely matter in practice.

remains a problem in the analysis of code, because of issues like polymorphism and aliasing. Section 4.2 and Section 5.4 discuss particularly notable approaches to handling them. Co-reference ambiguities can arise when viewing code statically, especially in dynamically typed languages (Section 6.1). The undefined behavior that some programming languages permit can cause semantic ambiguity and, in the field, syntactic problems can arise due to nonstandard compilers [24]; however, the dominance of a handful of compilers/interpreters for most languages ameliorates both problems.

Cross-Channel Interaction. Code's two channels, the algorithmic and the explanatory channels, interact through their semantic units, but mapping code units to textual units remains an open problem. Natural semantic units in code are identifiers, statements, blocks, and functions. None of these universally maps to textual semantic units. For example, identifiers, even verbose function names that seek to describe their function, carry less information than words like "Christmas" or "set". In general, statements in code and sentences in text differ in how much background knowledge the reader needs in order to understand them in isolation; an arbitrary statement is far more likely to use domain-specific, even project-specific, names or neologisms than an arbitrary sentence is. Blocks vary greatly in length and semantics richness and often lack clear boundaries to a human reader. Functions are clearly delimited and semantically rich, but long. In text, a sentence is the natural multiword semantic unit and usually contains fewer than 50 words (tokens). Unfortunately, one cannot, however, easily equate them. A function differs from a sentence or a sequence of sentences, *i.e.* a paragraph, in that it is named and called, while, in general settings, sentences or paragraphs rarely have names or are referred to elsewhere in a text. Further, a single function acts on the world, so it is like a single *action* sentence, but is usually much longer, often containing hundreds of tokens, and usually performs multiple actions, making a function closer to a sequence of sentences, or a paragraph, but paragraphs are rarely solely composed of action sentences.

Additionally, parse trees of sentences in text tend to be diverse, short, and shallow compared to abstract syntax trees of functions, which are usually much deeper with repetitive internal structure. Code bases are multilingual (*i.e.* contain code in more than one programming language, *e.g.* Java and SQL) with different tasks described in different languages, more frequently than text corpora; this can drastically change the shape and frequency of its semantic units. Code has a higher neologism rate than text. Almost 70% of all characters are identifiers and a developer must choose a name for each one [50]; when writing text, an author rarely names new things but usually chooses an existing word to use. Existing work handles code's neologism rate by introducing cache mechanisms or decomposing identifiers at a subtoken level (Section 4.1).

Determining which semantic code unit is most useful for which task is an open question. Consider the problem of automatically generating comments that describe code, which can be formalized as a machine translation problem from code to text. Statistical machine translation approaches learn from an aligned corpus. Statement-granular alignment yields redundant comments, while function granular alignment has saliency issues (Section 5.5). As another example, consider code search, where search engines must map queries into semantic code units. Perhaps the answer will be in maps from code to text whose units vary by granularity or context (Section 6.1).

4 PROBABILISTIC MODELS OF CODE

In this section, we turn our attention to probabilistic machine learning models of source code. A probabilistic model of source code is a probability distribution over code artifacts. As all models do, probabilistic machine learning models make simplifying assumptions about the modeled domain. These assumptions make the models tractable to learn and use, but introduce error. Since each model makes different assumptions, each model has its own strengths and weaknesses and is more suitable for some applications. In this section, we group existing work into families of models and

discuss their assumptions. To group these family of models in terms of shared design choices, we separate these models into three categories, based on the form of the equation of the modeled probability distribution and their inputs and outputs, with the caveat that some models fall into multiple categories. We also discuss how and why these models differ from their natural language counterparts. In Section 5, we discuss applications of these models in software engineering and programming languages.

Code-generating Models define a probability distribution over code by stochastically modeling the generation of smaller and simpler parts of code, *e.g.* tokens or AST nodes.

Representational Models of Code take an abstract representation[4] of code as input. Example representations include token contexts or data flow. The resulting model yields a conditional probability distribution over code element properties, like the types of variables, and can predict them.

Pattern Mining Models infer, without supervision, a likely latent structure within code. These models are an instantiation of clustering in the code domain; they can find reusable and human-interpretable patterns.

Code-generating models find analogues in generative models of text, such as language models and machine translation models. Code representational models are analogous to systems for named entity recognition, text classification, and sentiment analysis in NLP. Finally, code pattern mining models are analogous to probabilistic topic models and ML techniques for mining structured information (*e.g.* knowledge-bases) from text. To simplify notation below, we use $\mathbb{c}$ to denote an arbitrary code abstraction, like an AST.

4.1 Code-generating Probabilistic Models of Source Code

Code-generating probabilistic models of code are probability distributions that describe a stochastic process for generating valid code, *i.e.* they model *how* code is written. Given training data $\mathcal{D}$, an output code representation $\mathbb{c}$, and a possibly empty context $C(\mathbb{c})$, these models learn the probability distribution $P_{\mathcal{D}}(\mathbb{c}|C(\mathbb{c}))$ and sample $P_{\mathcal{D}}$ to generate code. When $C(\mathbb{c}) = \emptyset$, the probability distribution $P_{\mathcal{D}}$ is a *language model* of code, *i.e.* it models how code is generated when no external context information is available. When $C(\mathbb{c})$ is a non-code modality (*e.g.* natural language), $P_{\mathcal{D}}$ describes a *code-generative multimodal model* of code. When $C(\mathbb{c})$ is also code, the probability distribution $P_{\mathcal{D}}$ is a *transducer model* of code. In addition to generating code, by definition, code generating probabilistic models act as a *scoring function*, assigning a non-zero probability to every possible snippet of code. This score, sometimes referred to as "naturalness" of the code [87], suggests how probable the code is under a learned model.

Since code-generating models predict the complex structure of code, they make simplifying assumptions about the generative process and iteratively predict elements of the code to generate a full code unit, *e.g.* code file or method. Because of code's structural complexity and the simplifying assumptions these models make to cope with it, none of the existing models in the literature generate code that *always* parses, compiles, and executes. Some of the models do, however, impose constraints that take code structure into account to remove some inconsistencies; for instance, [126] only generate variables declared within each scope.

We structure the discussion about code-generating models of code as follows. We first discuss how models in this category generate code and then we show how the three different types of models (language, transducer and multimodal models) differ.

[4] In the machine learning literature, *representation*, applied to code, is roughly equivalent to *abstraction* in programming language research: a lossy encoding that preserves a semantic property of interest.

4.1.1 Representing Code in Code-Generating Models. Probabilistic models for generating structured objects are widely in use in machine learning and natural language processing with a wide range of applications. Machine learning research is considering a wide range of structures from natural language sentences to chemical structures and images. Within the source code domain, we can broadly find three categories of models based on the way they generate code's structure: token-level models that generate code as a sequence of tokens, syntactic models that generate code as a tree and semantic models that generate graph structures. Note that this distinction is about the generative process and *not* about the information used within this process. For example Nguyen et al. [144] uses syntactic context, but is classified as a token-level model that generates tokens.

Token-level Models (sequences). Sequence-based models are commonly used because of their simplicity. They view code as a sequence of elements, usually code tokens or characters, *i.e.* $\mathbb{c} = t_1 \dots t_M$. Predicting a large sequence in a single step is infeasible due to the exponential number of possible sequences; for a set of V elements, there are $|V|^N$ sequences of length N. Therefore, most sequence-based models predict sequences by sequentially generating each element, *i.e.* they model the probability distribution $P(t_m|t_1 \dots t_{m-1}, C(\mathbb{c}))$. However, directly modeling this distribution is impractical and all models make different simplifying assumptions.

The n-gram model has been a widely used sequence-based model, most commonly used as a language model. It is an effective and practical LM for capturing local and simple statistical dependencies in sequences. n-gram models assume that tokens are generated sequentially, left-to-right and that the next token can be predicted using only the previous $n-1$ tokens. The consequence of capturing a short context is that n-gram models cannot handle long-range dependencies, notably scoping information. Formally, the probability of a token t_m, is conditioned on the context $C(\mathbb{c})$ (if any) and the generated sequence so far $t_1 \dots t_{m-1}$, which is assumed to depend on only the previous $n-1$ tokens. Under this assumption, we write

$$P_{\mathcal{D}}(\mathbb{c}|C(\mathbb{c})) = P(t_1 \dots t_M|C(\mathbb{c})) = \prod_{m=1}^{M} P(t_m|t_{m-1} \dots t_{m-n+1}, C(\mathbb{c})). \tag{1}$$

To use this equation, we need to know the conditional probabilities $P(t_m|t_{m-1} \dots t_{m-n+1}, C(\mathbb{c}))$ for each possible n-gram and context. This is a table of $|V|^n$ numbers for each context $C(\mathbb{c})$. These are the *parameters* of the model that we learn from the training corpus. The simplest way to estimate the model parameters is to set $P(t_m|t_{m-1} \dots t_{m-n+1})$ to the proportion of times that t_m follows $t_{m-1} \dots t_{m-n+1}$. In practice, this simple estimator does not work well, because it assigns zero probability to n-grams that do not occur in the training corpus. Instead, n-gram models use *smoothing* methods [40] as a principled way for assigning probability to unseen n-grams by extrapolating information from m-grams ($m < n$). Furthermore, considering n-gram models with non-empty contexts $C(\mathbb{c})$ exacerbates sparsity rendering these models impractical. Because of this, n-grams are predominantly used as language models. The use of n-gram LMs in software engineering originated with the pioneering work of Hindle et al. [88] who used an n-gram LM with Kneser-Ney [104] smoothing. Most subsequent research has followed this practice.

In contrast to text, code tends to be more verbose [88] and much information is lost within the $n-1$ tokens of the context. To tackle this problem, Nguyen et al. [144] extended the standard n-gram model by annotating the code tokens with parse information that can be extracted from the currently generated sequence. This increases the available context information allowing the n-gram model to achieve better predictive performance. Following this trend, but using concrete and abstract semantics of code, Raychev et al. [166] create a token-level model that treats code generation as a combined synthesis and probabilistic modeling task.

Tu et al. [180] and later, Hellendoorn and Devanbu [84] noticed that code has a high degree of *localness*, where identifiers (*e.g.* variable names) are repeated often within close distance. In their work, they adapted work in speech and natural language processing [109] adding a cache mechanism that assigns higher probability to tokens that have been observed most recently, achieving significantly better performance compared to other *n*-gram models. Modeling identifiers in code is challenging [6, 11, 29, 126]. The agglutinations of multiple subtokens (*e.g.* in `getFinalResults`) when creating identifiers is one reason. Following recent NLP work that models subword structure (*e.g.* morphology) [174], explicitly modeling subtoken in identifiers may improve the performance of generative models. Existing token-level code-generating models do not produce syntactically valid code. Raychev et al. [166] added additional context in the form of constraints — derived from program analysis — to avoid generating some incorrect code.

More recently, sequence-based code models have turned to deep recurrent neural network (RNN) models to outperform *n*-grams. These models predict each token sequentially, but loosen the fixed-context-size assumption, instead representing the context using a distributed vector representation (Section 4.2). Following this trend, Karpathy et al. [103] and Cummins et al. [48] use character-level LSTMs [91]. Similarly, White et al. [188] and Dam et al. [49] use token-level RNNs. Recently, Bhoopchand et al. [26] used a token sparse pointer-based neural model of Python that learns to copy recently declared identifiers to capture very long-range dependencies of identifiers, outperforming standard LSTM models[5].

Although neural models usually have superior predictive performance, training them is significantly more costly compared to *n*-gram models usually requiring orders of magnitude more data. Intuitively, there are two reasons why deep learning methods have proven successful for language models. First, the hidden state in an RNN can encode longer-range dependencies of variable-length beyond the short context of *n*-gram models. Second, RNN language models can learn a much richer notion of similarity across contexts. For example, consider an 13-gram model over code, in which we are trying to estimate the distribution following the context `for(int i=N; i>=0; i--)`. In a corpus, few examples of this pattern may exist because such long contexts occur rarely. A simple *n*-gram model cannot exploit the fact that this context is very similar to `for(int j=M; j>=0; j--)`. But a neural network *can* exploit it, by learning to assign these two sequences similar vectors.

Syntactic Models (trees). Syntactic (or structural) code-generating models model code at the level of abstract syntax trees (ASTs). Thus, in contrast to sequence-based models, they describe a stochastic process of generating tree structures. Such models make simplifying assumptions about how a tree is generated, usually following generative NLP models of syntactic trees: they start from a root node, then sequentially generate children top-to-bottom and left-to-right. Syntactic models generate a tree node conditioned on context defined as the forest of subtrees generated so far. In contrast to sequence models, these models — by construction — generate syntactically correct code. In general, learning models that generate tree structures is harder compared to generating sequences: it is relatively computationally expensive, especially for neural models, given the variable shape and size of the trees that inhibit efficient batching. In contrast to their wide application in NLP, probabilistic context free grammars (PCFG) have been found to be unsuitable as language models of code [126, 164]. This may seem surprising, because most parsers assume that programming languages are context-free. But the problem is that the PCFGs are not a good model of *statistical* dependencies between code tokens, because nearby tokens may be far away in the AST. So it is not that PCFGs do not capture long-range dependencies (*n*-gram-based models do not

[5]This work differs from the rest from the fact that it anonymizes/normalizes identifiers, creating a less sparse problem. Because of the anonymization, the results are not directly comparable with other models.

either), but that they do not even capture close-range dependencies that matter [29]. Further, ASTs tend to be deeper and wider than text parse trees due to the highly compositional nature of code.

Maddison and Tarlow [126] and Allamanis et al. [13] increase the size of the context considered by creating a non-context-free log-bilinear neural network grammar, using a distributed vector representation for the context. Additionally, Maddison and Tarlow [126] restricts the generation to generate variables that have been declared. To achieve this, they use the deterministically-known information and filter out invalid output. This simple process always produces correct code, even when the network does not learn to produce it. In contrast, Amodio et al. [16] create a significantly more complex model that aims to learn to enforce deterministic constraints of the code generation, rather than enforcing them on the directly on the output. We further discuss the issue of embedding constraints and problem structure in models *vs.* learning the constraints in Section 6.

Bielik et al. [29], Raychev et al. [164] increase the context by annotating PCFGs with a learned program that uses features from the code. Although the programs can, in principle, be arbitrary, they limit themselves to synthesizing decision tree programs. Similarly, Wang et al. [185], Yin and Neubig [196] use an LSTM over AST nodes to achieve the same goal. Allamanis and Sutton [12] also create a syntactic model learning Bayesian TSGs [43, 156] (see Section 4.3).

Semantic Models (graphs). Semantic code-generating models view code as a graph. Graphs are a natural representation of source code that require little abstraction or projection. Therefore, graph model can be thought as generalizations of sequence and tree models. However, generating complex graphs is hard, since there is no natural "starting" point or generative process, as reflected by the limited number of graphs models in the literature. We refer the interested reader to the related work section of Johnson [98] for a discussion of recent models in the machine learning literature. To our knowledge, there are no generative models that directly generate graph representations of realistic code (*e.g.* data-flow graphs). Nguyen and Nguyen [139] propose a generative model, related to graph generative models in NLP, that suggests application programming interface (API) completions. They train their model over API usages. However, they predict entire graphs as completions and perform no smoothing, so their model will assign zero probability to unseen graphs. In this way, their model differs from graph generating models in NLP, which can generate arbitrary graphs.

4.1.2 *Types of Code Generating Models.*

We use external context $C(\mathbb{c})$ to refine code generating models into three subcategories.

Language Models. Language models model the language itself, without using any external context, *i.e.* $C(\mathbb{c}) = \varnothing$. Although LMs learn the high-level structure and constraints of programming languages fairly easily, predicting and generating source code identifiers (*e.g.* variable and method names), long-range dependencies and taking into account code semantics makes the language modeling of code a hard and interesting research area. We discuss these and other differences and their implications for probabilistic modeling in Section 6.

Code LMs are evaluated like LMs in NLP, using perplexity (or equivalently cross-entropy) and word error rate. Cross-entropy H is the most common measure. Language models — as most predictive machine learning models — can be seen as compression algorithms where the model predicts the full output (*i.e.* decompresses) using extra information. Cross-entropy measures the average number of extra bits of information per token of code that a model needs to decompress the correct output using a perfect code (in the information-theoretic sense)

$$H(\mathbb{c}, P_{\mathcal{D}}) = -\frac{1}{M} \log_2 P_{\mathcal{D}}(\mathbb{c}) \tag{2}$$

where M is the number of tokens within $\mathbb{c}$. By convention, the average is reported per-token, even for non-token models. Thus, a "perfect" model, correctly predicting all tokens with probability 1, would require no additional bits of information because, in a sense, already "knows" everything. Cross-entropy allows comparisons across different models. Other, application-specific measures, are used when the LM was trained for a specific task, such as code completion (Section 5.1).

Code Transducer Models. Inspired by statistical machine translation (SMT), transducer models translate/transduce code from one format into another (*i.e.* $C(\mathbb{c})$ is also code), such as translating code from one source language into another, target language. They have the form $P_{\mathcal{D}}(\mathbb{c}|\mathbb{s})$, where $\mathbb{c}$ is the target source code that is generated and $C(\mathbb{c}) = \mathbb{s}$ is the source source code. Most code transducer models use phrase-based machine translation. Intuitively, phrase-based models assume that small chunks from the source input can directly be mapped to chunks in the output. Although this assumption is reasonable in NLP and many source code tasks, these models present challenges in capturing long-range dependencies within the source and target. For example, as we will mention in the next section, transducing code from an imperative source language to a functional target is not currently possible because of the source and target are related with a significantly more complicated relation that simply matching "chunks" of the input code to "chunks" in the output.

These types of models have found application within code migration [2, 102, 141], pseudocode generation [146] and code fixing [160]. Traditionally transducer models have followed a noisy channel model, in which they combine a language model $P_{\mathcal{D}}(\mathbb{c})$ of the target language with a translation/transduction model $P_{\mathcal{D}}(\mathbb{s}|\mathbb{c})$ to match elements between the source and the target. These methods pick the optimal transduction c^* such that $\mathbb{c}^* = \arg\max P_{\mathcal{D}}(\mathbb{c}|\mathbb{s}) = \arg\max P_{\mathcal{D}}(\mathbb{s}|\mathbb{c})P_{\mathcal{D}}(\mathbb{c})$, where the second equality derives from the Bayes equation. Again, these probabilistic generative models of code do *not* necessarily produce valid code, due to the simplifying assumptions they make in both $P_{\mathcal{D}}(\mathbb{c})$ and $P_{\mathcal{D}}(\mathbb{s}|\mathbb{c})$. More recently, machine translation methods based on phrase-based models and the noisy channel model have been outperformed by neural network-based methods that directly model $P_{\mathcal{D}}(\mathbb{c}|\mathbb{s})$.

Transducer models can be evaluated with SMT evaluation measures, such as BLEU [150] — commonly used in machine translation as an approximate measure of translation quality — or programming and logic-related measures (*e.g. "Does the translated code parse/compile?"* and *"Are the two snippets equivalent?"*).

Multimodal Models. Code-generating multimodal models correlate code with one or more non-code modalities, such as comments, specifications, or search queries. These models have the form $P_{\mathcal{D}}(\mathbb{c}|\mathbf{m})$ *i.e.* $C(\mathbb{c}) = \mathbf{m}$ is a representation of one or more non-code modalities. Multimodal models are closely related to representational models (discussed in Section 4.2): multimodal code-generating models learn an intermediate representation of the non-code modalities $\mathbf{m}$ and use it to generate code. In contrast, code representational models create an intermediate representation of the code but are *not* concerned with code generation.

Multimodal models of code have been used for code synthesis, where the non-code modalities are leveraged to estimated a conditional generative model of code, *e.g.* synthesis of code given a natural language description by Gulwani and Marron [77] and more recently by Yin and Neubig [196]. The latter model is a syntactic model that accepts natural language. Recently, Beltramelli [23], Deng et al. [51], Ellis et al. [55] designed multimodal model that accept visual input (the non-code modality) and generate code in a DSL describing how the input (hand-drawn image, GUI screenshot) was constructed. Another use of these models is to score the co-appearance of the modalities, *e.g.* in code search, to score the probability of some text given a textual query [13]. This stream of research is related to work in NLP and computer vision where one seeks to generate a natural language

description for an image. These models are closely related to the other code generating models, since they generate code. These models make also assume that the input modality conditions the generation process. Multimodal models combine an assumption with a design choice. Like language models, these models assume that probabilistic models can capture the process by which developers generate code; unlike language models, they additionally bias code generation using information from the input modality $\mathbf{m}$. The design choice is how to transform the input modality into an intermediate representation. For example, Allamanis et al. [13] use a bag-of-words assumption averaging the words' distributed representations. However, this limits the expressivity of the models because the input modality has to fit in whole within the distributed representation. To address this issue, Ling et al. [120] and Yin and Neubig [196] use neural attention mechanisms to selectively attend to information within the input modality without the need to "squash" all the information into a single representation. Finally, the text-to-code problem, in which we take the input modality $\mathbf{m}$ to be natural language text and the other modality $\mathbb{c}$ to be code, is closely related to the problem of semantic parsing in NLP; see Section 5.5.

4.2 Representational Models of Source Code

Generative models recapitulate the process of generating source code, but cannot explicitly predict facts about the code that may be directly useful to engineers or useful for other downstream tasks, such as static analyses. To solve this problem, researchers have built models to learn intermediate, not necessarily human-interpretable, encodings of code, like a vector embedding. These models predict the probability distribution of properties of code snippets, like variable types. We call them representational code models. They learn the conditional probability distribution of a code property π as $P_{\mathcal{D}}(\pi|f(\mathbb{c}))$, where f is a function that transforms the code $\mathbb{c}$ into a target representation and π can be an arbitrary set of features or other (variable) structures. These models use a diverse set of machine learning methods and are often application-specific. Table 2 lists representational code model research. Below we discuss two types of models. Note that they are *not* mutually exclusive and models frequently combine distributed representations and structured prediction.

4.2.1 Distributed Representations. Distributed representations [89] are widely used in NLP to encode natural language elements. For example, Mikolov et al. [131] learn distributed representations of words, showing that such representations can learn useful semantic relationships and Le and Mikolov [112] extend this idea to sentences and documents. Distributed representations refer to arithmetic vectors or matrices where the meaning of an element is *distributed* across multiple components (*e.g.* the "meaning" of a vector is distributed in its components). This contrasts with local representations, where each element is uniquely represented with exactly one component. Distributed representations are commonly used in machine learning and NLP because they tend to generalize better and have recently become extremely common due to their omnipresence in deep learning. Models that learn distributed representations assume that the elements being represented and their relations can be encoded within a multidimensional real-valued space and that the relation (*e.g.* similarity) between two representations can be measured within this space. Probabilistic code models widely use distributed representations. For example, models that use distributed vector representations learn a function of the form $\mathbb{c} \rightarrow \mathbb{R}^D$ that maps code elements to a D-dimensional vector. Such representations are usually the (learned) inputs or output of (deep) neural networks.

Allamanis et al. [6] learn distributed vector representations for variable and methods usage contexts and use them to predict a probability distribution over their names. Such distributed representations are quite similar to those produced by word2vec [131]; the authors found that the distributed vector representations of variables and methods learn common semantic properties, implying that some form of the distributional hypothesis in NLP also holds for code.

Table 1. Research on Source Code Generating Models $P_{\mathcal{D}}(\mathbb{c}|C(\mathbb{c}))$ (sorted alphabetically), describing the process of generating source code. References annotated with * are also included in other categories.

Reference	Type	Representation	$P_{\mathcal{D}}$	Application
Aggarwal et al. [2]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Token	Phrase	Migration
Allamanis and Sutton [11]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	—
Allamanis et al. [5]	$P_{\mathcal{D}}(\mathbb{c})$	Token + Location	n-gram	Coding Conventions
Allamanis and Sutton [12]*	$P_{\mathcal{D}}(\mathbb{c})$	Syntax	Grammar (pTSG)	—
Allamanis et al. [13]*	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Syntax	Grammar (NN-LBL)	Code Search/Synthesis
Amodio et al. [16]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax + Constraints	RNN	—
Barone and Sennrich [21]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Token	Neural SMT	Documentation
Beltramelli [23]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Token	NN (Encoder-Decoder)	GUI Code Synthesis
Bhatia and Singh [25]	$P_{\mathcal{D}}(\mathbb{c})$	Token	RNN (LSTM)	Syntax Error Correction
Bhoopchand et al. [26]	$P_{\mathcal{D}}(\mathbb{c})$	Token	NN (Pointer Net)	Code Completion
Bielik et al. [29]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax	PCFG + annotations	Code Completion
Campbell et al. [35]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	Syntax Error Detection
Cerulo et al. [37]	$P_{\mathcal{D}}(\mathbb{c})$	Token	Graphical Model (HMM)	Information Extraction
Cummins et al. [48]	$P_{\mathcal{D}}(\mathbb{c})$	Character	RNN (LSTM)	Benchmark Synthesis
Dam et al. [49]	$P_{\mathcal{D}}(\mathbb{c})$	Token	RNN (LSTM)	—
Gulwani and Marron [77]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Syntax	Phrase Model	Text-to-Code
Gvero and Kuncak [82]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax	PCFG + Search	Code Synthesis
Hellendoorn et al. [85]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	Code Review
Hellendoorn and Devanbu [84]	$P_{\mathcal{D}}(\mathbb{c})$	token	n-gram (cache)	–
Hindle et al. [88]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	Code Completion
Hsiao et al. [93]	$P_{\mathcal{D}}(\mathbb{c})$	PDG	n-gram	Program Analysis
Lin et al. [118]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Tokens	NN (Seq2seq)	Synthesis
Ling et al. [120]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Token	RNN + Attention	Code Synthesis
Liu [121]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	Obfuscation
Karaivanov et al. [102]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Token	Phrase	Migration
Karpathy et al. [103]	$P_{\mathcal{D}}(\mathbb{c})$	Characters	RNN (LSTM)	—
Kushman and Barzilay [110]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Token	Grammar (CCG)	Code Synthesis
Maddison and Tarlow [126]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax with scope	NN	—
Menon et al. [129]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Syntax	PCFG + annotations	Code Synthesis
Nguyen et al. [140]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Token	Phrase	Migration
Nguyen et al. [144]	$P_{\mathcal{D}}(\mathbb{c})$	Token + parse info	n-gram	Code Completion
Nguyen et al. [141]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Token + parse info	Phrase SMT	Migration
Nguyen and Nguyen [139]	$P_{\mathcal{D}}(\mathbb{c})$	Partial PDG	n-gram	Code Completion
Oda et al. [146]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Syntax + Token	Tree-to-String + Phrase	Pseudocode Generation
Patra and Pradel [153]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax	Annotated PCFG	Fuzz Testing
Pham et al. [154]	$P_{\mathcal{D}}(\mathbb{c})$	Bytecode	Graphical Model (HMM)	Code Completion
Pu et al. [160]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Token	NN (Seq2seq)	Code Fixing
Rabinovich et al. [162]*	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Syntax	NN (LSTM-based)	Code Synthesis
Raychev et al. [166]	$P_{\mathcal{D}}(\mathbb{c})$	Token + Constraints	n-gram/ RNN	Code Completion
Ray et al. [163]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram (cache)	Bug Detection
Raychev et al. [164]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax	PCFG + annotations	Code Completion
Saraiva et al. [173]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	—
Sharma et al. [175]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	Information Extraction
Tu et al. [180]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram (cache)	Code Completion
Vasilescu et al. [181]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbb{s})$	Token	Phrase SMT	Deobfuscation
Wang et al. [185]	$P_{\mathcal{D}}(\mathbb{c})$	Syntax	NN (LSTM)	Code Completion
White et al. [188]	$P_{\mathcal{D}}(\mathbb{c})$	Token	NN (RNN)	—
Yadid and Yahav [194]	$P_{\mathcal{D}}(\mathbb{c})$	Token	n-gram	Information Extraction
Yin and Neubig [196]	$P_{\mathcal{D}}(\mathbb{c}\|\mathbf{m})$	Syntax	NN (Seq2seq)	Synthesis

Abbreviations: • pTSG: probabilistic tree substitution grammar • NN: neural network • LBL: log-bilinear • SMT: statitstical machine translation • PCFG: probabilistic context-free grammar • HMM: hidden Markov model • LSTM: long short-term memory • RNN: recurrent neural network • CCG: combinatory categorial grammar • Seq2seq: sequence-to-sequence neural network

Gu et al. [76] use a sequence-to-sequence deep neural network [177], originally introduced for SMT, to learn intermediate distributed vector representations of natural language queries which they use to predict relevant API sequences. Mou et al. [132] learn distributed vector representations using custom convolutional neural networks to represent features of snippets of code, then they assume that student solutions to various coursework problems have been intermixed and seek to recover the solution-to-problem mapping via classification.

Li et al. [115] learn distributed vector representations for the nodes of a memory heap and use the learned representations to synthesize candidate formal specifications for the code that produced the heap. Li et al. [115] exploit heap structure to define graph neural networks, a new machine learning model based on gated recurrent units (GRU, a type of RNN [41]) to directly learn from heap graphs. Piech et al. [155] and Parisotto et al. [151] learn distributed representations of source code input/output pairs and use them to assess and review student assignments or to guide program synthesis from examples.

Neural code-generative models of code also use distributed representations to capture context, a common practice in NLP. For example, the work of Maddison and Tarlow [126] and other neural language models (*e.g.* LSTMs in Dam et al. [49]) describe context distributed representations while sequentially generating code. Ling et al. [120] and Allamanis et al. [13] combine the code-context distributed representation with a distributed representations of other modalities (*e.g.* natural language) to synthesize code. While all of these representations can, in principle, encode unbounded context, handling all code dependencies of arbitrary length is an unsolved problem. Some neural architectures, such as LSTMs [91], GRUs [41] and their variants, have made progress on this problem and can handle moderately long-range dependencies.

4.2.2 Structured Prediction. Structured prediction is the problem of predicting a set of interdependent variables, given a vector of input features. Essentially, structured prediction generalizes standard classification to multiple output variables. A simple example of structured prediction is to predict a part-of-speech tag for each word in a sentence. Often the practitioner defines a dependency structure among the outputs, *e.g.*, via a graph, as part of the model definition. Structured prediction has been widely studied within machine learning and NLP, and are omnipresent in code. Indeed, structured prediction is particularly well-suited to code, because it can exploit the semantic and syntactic structure of code to define the model. Structured prediction is a general framework to which deep learning methods have been applied. For example, the celebrated sequence-to-sequence (seq2seq) learning models [19, 177] are general methods for tackling the related structured prediction problem. In short, structured prediction and distributed representations are not mutually exclusive.

One of the most well-known applications of structured prediction to source code is Raychev et al. [165], who represent code as a variable dependency network, represent each JavaScript variable as a single node, and model their pairwise interactions as a conditional random field (CRF). They train the CRF to jointly predict the types and names of all variables within a snippet of code. Proksch et al. [159] use a directed graphical model to represent the context of an (incomplete) usage of an object to suggest a method invocation (*viz.* constructor) autocompletion in Java.

Structured prediction, such as predicting a sequence of elements, can be combined with distributed representations. For example, Allamanis et al. [6, 10] use distributed representations to predict sequences of identifier sub-tokens to build a single token and Gu et al. [76] predict the sequence of API calls. Li et al. [115] learn distributed representations for the nodes of a fixed heap graph by considering its structure and the interdependencies among the nodes. Kremenek et al. [108] use a factor graph to learn and enforce API protocols, like the resource usage specification of the POSIX file API, as do Livshits et al. [122] for information flow problems. Allamanis et al. [8] predict the

Table 2. Research on Representational Models of Source Code $P_{\mathcal{D}}(\pi|f(\mathbb{c}))$ (sorted alphabetically). References annotated with * are also included in other categories. GM refers to graphical models

Reference	Input Code Representation ($\mathbb{c}$)	Target (π)	Intermediate Representation (f)	Application
Allamanis et al. [6]	Token Context	Identifier Name	Distributed	Naming
Allamanis et al. [13]*	Natural Language	LM (Syntax)	Distributed	Code Search
Allamanis et al. [10]	Tokens	Method Name	Distributed	Naming
Allamanis et al. [8]	PDG	Variable Use Bugs	Distributed	Program Analysis
Bavishi et al. [22]	Token Context	Identifier Name	Distributed	Naming
Bichsel et al. [27]	Dependency Net	Identifier Name	CRF (GM)	Deobfuscation
Bruch et al. [33]	Partial Object Use	Invoked Method	Localized	Code Completion
Chae et al. [38]	Data Flow Graph	Static Analysis	Localized	Program Analysis
Corley et al. [44]	Tokens	Feature Location	Distributed	Feature Location
Cummins et al. [47]	Tokens	Optimization Flags	Distributed	Optimization Heuristics
Dam et al. [49]*	Token Context	LM (Tokens)	Distributed	—
Gu et al. [76]	Natural Language	API Calls	Distributed	API Search
Guo et al. [79]	Tokens	Traceability link	Distributed	Traceability
Gupta et al. [81]	Tokens	Code Fix	Distributed	Code Fixing
Gupta et al. [80]	Tokens	Code Fix	Distributed	Code Fixing
Hu et al. [94]	Linearized AST	Natural Language	Distributed	Summarization
Iyer et al. [96]	Tokens	Natural Language	Distributed	Summarization
Jiang et al. [97]	Tokens (Diff)	Natural Language	Distributed	Commit Message
Koc et al. [106]	Bytecode	False Positives	Distributed	Program Analysis
Kremenek et al. [108]	Partial PDG	Ownership	Factor (GM)	Pointer Ownership
Levy and Wolf [114]	Statements	Alignment	Distributed	Decompiling
Li et al. [115]	Memory Heap	Separation Logic	Distributed	Verification
Loyola et al. [124]	Tokens (Diff)	Natural Language	Distributed	Explain code changes
Maddison and Tarlow [126]*	LM AST Context	LM (AST)	Distributed	—
Mangal et al. [127]	Logic + Feedback	Prob. Analysis	MaxSAT	Program Analysis
Movshovitz-Attias and Cohen [133]	Tokens	Code Comments	Directed GM	Comment Prediction
Mou et al. [132]	Syntax	Classification	Distributed	Task Classification
Nguyen et al. [142]	API Calls	API Calls	Distributed	Migration
Omar [148]	Syntactic Context	Expressions	Directed GM	Code Completion
Oh et al. [147]	Features	Analysis Params	Static Analysis	Program Analysis
Piech et al. [155]	Syntax + State	Student Feedback	Distributed	Student Feedback
Pradel and Sen [157]	Syntax	Bug Detection	Distributed	Program Analysis
Proksch et al. [159]	Inc. Object Usage	Object Usage	Directed GM	Code Completion
Rabinovich et al. [162]*	LM AST Context	LM (AST)	Distributed	Code Synthesis
Raychev et al. [165]	Dependency Net	Types + Names	CRF (GM)	Types + Names
Wang et al. [183]	Tokens	Defects	LM (n-gram)	Bug Detection
White et al. [188]*	Tokens	LM (Tokens)	Distributed	—
White et al. [187]*	Token + AST	—	Distributed	Clone Detection
Zaremba and Sutskever [197]	Characters	Execution Trace	Distributed	—

data flow graph of code by learning to paste snippets of code into existing code and adapting the variables used.

4.3 Pattern Mining Models of Source Code

Pattern mining models aim to discover a finite set of human-interpretable patterns from source code, without annotation or supervision, and present the mined patterns to software engineers. Broadly, these models cluster source code into a finite set of groups. Probabilistic pattern mining models of code infer the likely latent structure of a probability distribution

$$P_{\mathcal{D}}(f(\mathbb{c})) = \sum_{\mathbb{l}} P_{\mathcal{D}}(g(\mathbb{c})|\mathbb{l})P(\mathbb{l}) \tag{3}$$

Table 3. Research on Pattern Mining Probabilistic Models of Source Code (sorted alphabetically). These models have the general form $P_{\mathcal{D}}(g(\mathbb{c}))$. References annotated with * are also included in other categories.

Reference	Code Representation ($\mathbb{c}$)	Representation (g)	Application
Allamanis and Sutton [12]*	Syntax	Graphical Model	Idiom Mining
Allamanis et al. [4]	Abstracted AST	Graphical Model	Semantic Idiom Mining
Fowkes and Sutton [63]	API Call Sequences	Graphical Model	API Mining
Murali et al. [135]	Sketch Synthesis	Graphical Model	Sketch Mining
Murali et al. [136]	API Usage Errors	Graphical Model	Defect Prediction
Movshovitz-Attias and Cohen [134]	Tokens	Graphical Model	Knowledge-Base Mining
Nguyen et al. [143]	API Usage	Distributed	API Mining
Fowkes et al. [62]	Tokens	Graphical Model	Code Summarization
Wang et al. [184]	Serialized ASTs	Distributed	Defect Prediction
White et al. [187]*	Token & Syntax	Distributed	Clone Detection

where g is a deterministic function that returns a (possibly partial, *e.g.* API calls only) view of the code and $\mathbf{l}$ represents a set of latent variables that the model introduces and aims to infer. Applications of such models are common in the mining software repositories community and include documentation (*e.g.* API patterns), summarization, and anomaly detection. Table 3 lists this work. Unsupervised learning is one of the most challenging areas in machine learning. This hardness stems from the need to automatically distinguish important patterns in the code from spurious patterns that may appear to be significant because of limited and noisy data. When designing unsupervised models, the core assumption lies in the objective function being used and often we resort to using a principle from statistics, information theory or a proxy supervised task. Like all machine learning models, they require assumptions about how the data is represented. An important issue with unsupervised methods is the hardness of evaluating the output, since the quality of the output is rarely quantifiable. A vast literature on *non-probabilistic* methods exploits data mining methods, such as frequent pattern mining and anomaly detection [190]. We do *not* discuss these models here, since they are *not* probabilistic models of code. Classic probabilistic topic models [30], which usually views code (or other software engineering artifacts) as a bag-of-words, have also been heavily investigated. Since these models and their strengths and limitations are well-understood, we omit them here.

Allamanis and Sutton [12] learn a tree substitution grammar (TSG) using Bayesian nonparametrics, a technique originally devised for natural language grammars. TSGs learn to group commonly co-appearing grammar productions (tree fragments). Although TSGs have been used in NLP to improve parsing performance (which is ambiguous in text), Allamanis and Sutton [12] observe that the inferred fragments represent common code usage conventions and name them *idioms*. Later, Allamanis et al. [4] extend this technique to mine semantic code idioms by modifying the input code representation and adapting the inference method.

In a similar fashion, Fowkes and Sutton [63] learn the latent variables of a graphical model to infer common API usage patterns. Their method automatically infers the most probable grouping of API elements. This is in stark contrast to frequency-based methods [192] that suffer from finding frequent but not necessarily "interesting" patterns. Finally, Movshovitz-Attias and Cohen [134] infer the latent variables of a graphical model that models a software ontology.

As in NLP and machine learning in general, evaluating pattern mining models is hard, since the quality of the discovered latent structure is subjective. Thus, researchers often resort to extrinsic, application-specific measures. For example, Fowkes et al. [62] run a user study to directly assess the quality of their summarization method.

5 APPLICATIONS

Probabilistic models of source code have found a wide range of applications in software engineering and programming language research. These models enable the principled use of probabilistic reasoning to handle uncertainty. Common sources of uncertainty are underspecified or inherently ambiguous data (such as natural language text). In some domains, probabilistic source code models also simplify or accelerate analysis tasks that would otherwise be too computationally costly to execute. In this section, our goal is to explain the use of probabilistic models in each area, not review them in detail. We describe each area's goals and key problems, then explain how they can benefit from probabilistic, machine learning-based methods, and how the methods are evaluated.

5.1 Recommender Systems

Software engineering recommender systems [169, 170] make recommendations to assist software engineering tasks, such as code autocompletion and recommending likely code reviewers for a given code change. Many of these systems employ data mining and machine learning approaches on various software engineering artifacts. Probabilistic models of code find application in source code-based recommender systems [130], such as those that aid developers write or maintain code.

Modeling developer *intent* is a challenge: even if there were an agreed upon way to formalize intent, developers are reluctant to formalize their intent separately from their code itself. Probabilistic reasoning is well-suited for inferring intent, since it allows us to quantify the uncertainty that is inherent to inferring any latent variable. Probabilistic recommender systems extract information from the *context* of (partial) code and use it to probabilistically reason about developer intent.

The most prominent recommender system and a feature commonly used in integrated development environment (IDEs) is *code completion*. All widely used IDEs, such as Eclipse, IntelliJ and Visual Studio, have some code completion features. According to Amann et al. [14], code completion is the most used IDE feature. However, code completion tools typically return suggestions in alphabetic order, rather than in relative order of predicted relevance to the context. Statistical code completion aims to improve suggestion accuracy by learning probabilities over the suggestions and providing to the users a ranked list. Some systems focus on automatically completing specific constructs (*e.g.* method calls and parameters); others try to complete all code tokens. In all cases, probabilistic code completion systems use existing code as their training set.

Statistical code completion was first studied by Bruch et al. [33] who extracted features from code context to suggest completions for method invocations and constructors. Later, Proksch et al. [159] used Bayesian graphical models (structured prediction) to improve accuracy. This context-based model captures all usages of an object and models the probability distribution for the next call. A version of this research is integrated into the Eclipse IDE under Eclipse Recommenders [61].

Source code language models have implicitly and explicitly been used for code completion. Hindle et al. [88] were the first to use a token-level n-gram LM for this purpose, using the previous $n-1$ tokens to represent the completion context at each location. Later, Franks et al. [64], Tu et al. [180] used a cache n-gram LM and further improved the completion performance, showing that a local cache acts as a domain adapted n-gram. Nguyen et al. [144] augment the completion context with semantic information, improving the code completion accuracy of the n-gram LM. Raychev et al. [166] exploit formal properties of the code in context to limit incorrect (but statistically probable) API call suggestions. Their method is the first to depart from simple statistical token completion towards statistical program synthesis of single statements. Apart from token-level language models for code completion, Bielik et al. [29] and Maddison and Tarlow [126] create AST-level LMs that can be used for suggestion.

In contrast to work that predicts source code, Movshovitz-Attias and Cohen [133] create a recommender system to assist comment completion given a source code snippet, using a topic-like graphical model to model context information. Similarly, the work of Allamanis et al. [5, 6, 10] can be seen as a recommender systems for suggesting names for variables, methods, and classes by using relevant code tokens as the context.

5.2 Inferring Coding Conventions

Coding conventions are syntactic constraints on code beyond those imposed by the grammar of a programming language. They govern choices like formatting (brace and newline placement) or Hungarian notation *vs.* CamelCase naming. They seek to prevent some classes of bugs and make code easier to comprehend, navigate, and maintain [5]. In massive, open, online courses, coding conventions help teachers identify and understand common student errors [71]. Enforcing coding conventions is tedious. Worse, it is sometimes difficult to achieve consensus on what they should be, a prerequisite for their codification in rule-based systems. Inferring coding conventions with machine learning solves this problem by *learning* emergent conventions directly from a codebase. This can help software teams to determine the coding conventions a codebase uses without the need to define rules rule upfront or configure existing convention enforcing tools.

Machine learning models of source code that look at the surface structure (*e.g.* tokens, syntax) are inherently well-suited for this task. Using the source code as data, they can infer the emergent conventions while quantifying uncertainty over those decisions. An important challenge in this application domain is the sparsity of the code constructs, caused by the diverse and non-repeatable form of source code within projects and domains. Allamanis et al. [5, 6, 10], Bavishi et al. [22] exploit the statistical similarities of code's surface structure to learn and suggest variable, method, and class naming conventions, while Allamanis and Sutton [12] and Allamanis et al. [4] mine conventional syntactic and semantic patterns of code constructs that they call *idioms*. They show that these idioms are useful for documentation and can help software engineering tool designers achieve better coverage of their tools. To handle code formatting conventions, Parr and Vinju [152] learn a source code formatter from data by using a set of hand-crafted features from the AST and a k-NN classifier.

5.3 Code Defects

Probabilistic models of source code assign high probability to code that appears often in practice, *i.e.* is natural. Therefore, code considered very improbable may be buggy. This is analogous to anomaly detection using machine learning [39]. Finding defects is a core problem in software engineering and programming language research. The challenge in this domain rests in correctly characterizing source code that contains a defects with high precision and recall. This is especially difficult because of the rarity of defects and the extreme diversity of (correct) source code.

Preliminary work suggests that the probability assigned by language models can indicate code defects. Allamanis and Sutton [11] suggest that n-gram LMs can be seen as complexity measures and Ray et al. [163] present evidence that buggy code tends to have lower probability (is less "natural") than correct code and show that LMs find defects as well as popular tools like FindBugs.

Wang et al. [184] use deep belief networks to automatically learn token-level source code features that predict code defects. Fast et al. [58] and Hsiao et al. [93] learn statistics from large numbers of code to detect potentially erroneous code and perform program analyses while Wang et al. [183] learn coarse-grained n-gram language models to detect uncommon usages of code. These models implicitly assume that a simple set of statistics or an LM can capture anomalous/unusual contexts. Recently, Murali et al. [136] use a combination of topic models to bias a recurrent neural network

that models the sequences of API calls in a learned probabilistic automaton. They use the model to detect highly improbable sequences of API calls detecting real-world bugs in Android code. Allamanis et al. [8], Pradel and Sen [157] use various elements from code context to detect specific kinds of bugs, such as variable and operator misuses.

Because of the sparsity of source code, work on detecting code defects uses different abstraction levels of source code. For example, Wang et al. [183] create coarse-grained *n*-grams, while Murali et al. [136] focus on possible paths (that remove control flow dependencies) over API calls. Therefore, each model captures a limited family of defects, determined by the model designers' choice of abstraction to represent. Pu et al. [160] and Gupta et al. [81] create models for detecting and fixing defects but only for student submissions where data sparsity is not a problem. Other data-mining based methods (*e.g.* Wasylkowski et al. [186]) also exist, but are out-of-scope from this review since they do not employ probabilistic methods.

Also related is the work of Campbell et al. [35] and Bhatia and Singh [25]. These researchers use source code LMs to identify and correct syntax errors. Detecting syntax errors is an easier and more well defined task. The goal of these models is *not* to detect the existence of such an error (that can be deterministically found) but to efficiently localize the error and suggest a fix.

The earlier work of Liblit et al. [117], Zheng et al. [199] use traces for statistical bug isolation. Kremenek et al. [108] learn factor graphs (structured prediction) to model resource-specific bugs by modeling resource usage specifications. These models use an efficient representation to capture bugs, but can fail on interprocedural code that requires more complex graph representations. Finally, Patra and Pradel [153] use an LM of source code to generate input for fuzz testing browsers.

Not all anomalous behavior is a bug (it may simply be rare behavior), but anomalous behavior in often executed code almost certainly is [56]. Thus, probabilistic models of source code seem a natural fit for finding defective code. They have not, however, seen much industrial uptake. One possible cause is their imprecision. The vast diversity of code constructs entails sparsity, from which all anomaly detection methods suffer. Methods based on probabilistic models are no exception: they tend to consider rare, but *correct*, code anomalous.

5.4 Code Translation, Copying, and Clones

The success of statistical machine translation (SMT) among (natural) languages has inspired researchers to use machine learning to translate code from one source language (*e.g.* Java) to another (*e.g.* C#). Although rule-based rewriting systems can be (and have been) used, it is tedious to create and maintain these rules, in the face of language evolution. SMT models are well suited for this task, although they tend to produce invalid code. To reduce these errors during translation Karaivanov et al. [102] and Nguyen et al. [141] add semantic constraints to the translation process.

Existing research has applied widely used SMT models for text. Although these models learn mappings between different language constructs such as APIs, they have only been used for translating between programming languages of similar paradigms and structure (C# and Java are both object-oriented languages with managed memory). This is an important limitation; machine learning innovations are required to translate between languages of different types (*e.g.* Java to Haskell or assembly to C) or even languages with different memory management (*e.g.* Java to C). Existing per-statement SMT from Java to C does not track memory allocations and therefore fails to emit memory de-allocations that C's lack of garbage collection requires. Similarly, translating object-oriented code to functional languages will require learning the conceptual differences of the two paradigms while preserving semantics, such as learning to translate a loop to a map-reduce functional. Researchers evaluate translation models by scoring exact matches, measuring

the syntactic or semantic correctness of the translated code, or using BLEU [150]. We discuss this and other measure-related issues in Section 6.

Developers often copy code during development. This practice requires fixups to rename variables and handle name collisions; it can also create code clones, similar code snippets in different locations of a code base [107]. Allamanis and Brockschmidt [7] automate naming cleanups after copying; they use structured prediction and distributed representations to adapt/port a pasted snippet's variables into the target context. Their method probabilistically represents semantic information about variable use to predict the correct name adaptations without external information (*e.g.* tests). Clones may indicate refactoring opportunities (that allow reusing the cloned code). White et al. [187] use autoencoders and recurrent neural networks [72] to find clones as code snippets that share similar distributed representations. Using distributed vector representations allows them to learn a continuous similarity metric between code locations, rather than using edit distance.

5.5 Code to Text and Text to Code

Linking natural language text to source code has many useful applications, such as program synthesis, traceability, search and documentation. However, the diversity of both text and code, the ambiguity of text, the compositional nature of code and the layered abstractions in software make interconnecting text and code a hard problem. Probabilistic machine learning models provide a principled method for modeling and resolving ambiguities in text and in code.

Generating natural language from source code, *i.e.* code-to-text, has applications to code documentation and readability. For example, Oda et al. [146] translate Python code to pseudocode (in natural language) using machine translation techniques, with the goal of producing a more readable generation of the code. Iyer et al. [96] design a neural attention model that summarizes code as text. Movshovitz-Attias and Cohen [133] generate comments from code using n-gram models and topic models.

The reverse direction, text-to-code, aims to help people, both developers and end users, write programs more easily. This area is closely related to semantic parsing in NLP. Semantic parsing is the task of converting a natural language utterance into a representation of its meaning, often database or logical queries that could subsequently be used for question answering [99]. We do not have space to fully describe the large body of work that has been done in semantic parsing in NLP, but instead will focus on text-to-code methods that output code in languages that are used by human software developers. This area has attracted growing interest, with applications such as converting natural language to Excel macros [77], Java expressions [82], shell commands [118, 119], simple if-then programs [161], regular expressions [110] and to SQL queries [200]. Finally, Yin and Neubig [196] have recently presented a neural architecture for general-purpose code generation. For more details, Neubig [138] provides an informal survey of code generation methods.

5.6 Documentation, Traceability and Information Retrieval

Improving documentation and code search are central questions in software engineering. Probabilistic models are particularly natural here because, as we have seen, they allow integrating information between NL text and code. Although the more general code-to-text and text-to-code models from the previous sections could clearly be applied here, researchers have often found, as in the NL domain, that more specialized solutions are currently effective for these problems.

Code search — a common activity for software engineers [14, 172] — can employ natural language queries. Software engineering researchers have focused on the code search problem using information retrieval (IR) methods [68, 92, 128, 178]. Niu et al. [145] has used learning-to-rank methods but with manually extracted features. Within the area of statistical models of source code,

Gu et al. [76] train a sequence-to-sequence (seq2seq) neural network to map natural language into API sequences. Allamanis et al. [13] learn a bimodal, generative model of code, conditioned on natural language text and use it to rank code search results. All of these methods use rank-based measures (*e.g.* mean reciprocal rank) to evaluate their performance.

Documentation is text that captures requirements, specifications, and descriptions of code. Engineers turn to it to prioritize new features and to understand code during maintenance. Searching, formalizing, reasoning about, and interlinking code to (*i.e.* the traceability problem of Gotel et al. [74]) documentation are seminal software engineering problems. Mining common API patterns is a recurring theme and there is a large literature of non-probabilistic methods (*e.g.* frequency-based) for mining and synthesizing API patterns [34, 192], which are out-of-scope of this review. Also out-of-scope is work that combines natural language information with APIs. For example, Treude and Robillard [179] extract phrases from StackOverflow using heuristics (manually selected regular expressions) and use off-the-self classifiers on a set of hand-crafted features. We refer the reader to Robillard et al. [169] for all probabilistic and non-probabilistic recommender systems. Within this domain, there are a few probabilistic code models that mine API sequences. Gu et al. [76] map natural language text to commonly used API sequences, Allamanis and Sutton [12] learn fine-grained source code idioms, that may include APIs. Fowkes and Sutton [63] uses a graphical model to mine interesting API sequences.

Documentation is also related to information extraction from (potentially unstructured) documents. Cerulo et al. [37] use a language model to detect code "islands" in free text. Sharma et al. [175] use a language model over tweets to identify software-relevant tweets.

5.7 Program Synthesis

Program synthesis is concerned with generating full or partial programs from a specification [78]. Traditionally, a specification is a formal statement in an appropriate logic. More recently, researchers have considered partial or incomplete specifications, such as input/output pairs [111] or a natural language description. Program synthesis generates full or partial programs from a specification. When the specification is a natural language description, this is the semantic parsing task (see Section 5.5). Program synthesis (*e.g.* from examples or a specification) has received a great deal of attention in programming language research. The core challenge is searching the vast space of possible programs to find one that complies with the specification. Probabilistic machine learning models help guiding the search process to more probable programs.

Research on programming by example (PBE) leverages machine learning methods to synthesize code. Liang et al. [116] use a graphical model to learn commonalities of programs across similar tasks with the aim to improve program synthesis search. Menon et al. [129] use features from the input/output examples to learn a parameterized PCFG to speed up synthesis. Singh and Gulwani [176] extract features from the synthesized program to learn a supervised classifier that can predict the correct program and use it to re-rank synthesis suggestions. The recent work of Balog et al. [20] and Parisotto et al. [151] combine ideas from existing enumerative search techniques with learned heuristics to learn to efficiently synthesize code, usually written within a DSL. Neelakantan et al. [137] and Reed and de Freitas [167] introduce neural differentiable architectures for program induction. This is an interesting emerging area of research [101], but does not yet scale to the types of problems considered by the programming language and software engineering community [59, 69]; also see Devlin et al. [53], Gaunt et al. [69] for a comparison of neural program induction and program synthesis methods. Finally, the code completion work of Raychev et al. [166] can be seen as a limited program synthesis of method invocations at specific locations.

Although program synthesis is usually referred in the context of generating a program that complies to some form of specification, probabilistic models have been used to synthesize random — but functioning — programs for benchmarks and compiler fuzzing. Cummins et al. [48] synthesize automatically a large number of OpenCL benchmarks by learning a character-level LSTM over valid OpenCL code. Their goal is to generate reasonable-looking code, rather than synthesize a program that complies with a specification. To ease their task, they normalize the code by consistently alpha-renaming variables and method names. Finally, they filter invalid intermediate output so they, in the end, generate only valid programs. In a similar manner, Patra and Pradel [153] synthesize JavaScript programs for fuzz testing JavaScript interpreters.

5.8 Program Analysis

Program analysis is an important area that seeks to soundly extract semantic properties, like correctness, from programs. Sound analyses, especially those that scale, can be imprecise and return unacceptably large numbers of false positives. Probabilistic models of code use probabilistic reasoning to alleviate these problems. Three distinct program analysis approaches have exploited probabilistic models of source code.

First, a family of models relaxes the soundness requirement, yielding probabilistic results instead. Raychev et al. [165] use a graphical model to predict the probability distribution of JavaScript variable types by learning statistical patterns of how code is used. Oh et al. [147] and Mangal et al. [127] use machine learning models to statistically parameterize program analyses to reduce false positive ratio while maintaining high precision. Chae et al. [38] reduce automatically (without machine learning) a program to a set of data-flow graphs, manually extract features from them. Using these features they then learn the appropriate parametrization of a static analysis, using a traditional classifier. Koc et al. [106] train a classifier, using LSTMs, to predict if a static analysis warning is a false positive. The neural network learns common patterns that can discriminate between false and true positives. The second paradigm that has been explored is to use machine learning to create models that produce plausible hypotheses of formal verification statements that can be proved. Brockschmidt et al. [31] and Li et al. [115] propose a set of models that generate separation logic expressions from the heap graph of a program, suitable for formally verifying its correctness. The third paradigm — although not yet applied directly to source code but to other forms of formal reasoning — learns heuristics to speed-up the search for a formal automated proof. The goal of such methods is to replace hard-coded heuristics with a learnable and adaptive module that can prioritize search tactics per-problem without human intervention. Alemi et al. [3] and Loos et al. [123] take a first step towards this direction by learning heuristic for automated theorem proving for mathematical expressions.

6 CHALLENGES AND FUTURE DIRECTIONS

The development and analysis of code must contend with uncertainty, in many forms: "*What is the purpose of this code?*", "*What does functionality does this unit test test?*", "*From this program, can we infer any of its specification?*", "*What is this program's intended (or likely) input domain?*" or "*Why did this program crash here?*". This contrasts with traditional program analysis which is conservative: it deems that a program has a property, like a bug, if that property is possible, independent of likelihood. In contrast, machine learning studies robust inference under uncertainty and noise. Thus, the application of machine learning to code and its development, is an emerging research topic with the potential to influence programming language and software engineering research. Here, we list topics where principled probabilistic reasoning promises new advances,

focusing on probabilistic models of code. We also list open challenges, some quite longstanding like long-range dependency, and speculate about directions for making progress on their resolution.

For each open problem, machine learning is introduced to handle uncertainty, ambiguity, or avoid hard-coded heuristics. Probabilistic learning systems model these as noise which they handle robustly by training statistical principled models. This in turn has allowed the creation of new previously impossible systems (*e.g.* text-to-code systems) or replaced existing, hard-coded heuristics with machine learning systems that promise to be more robust and generalize better. This course of direction highly resembles that of NLP, where hard-coded "expert" systems have been successfully replaced by sophisticated machine learning methods.

6.1 The Third Wave of Machine Learning

The first wave of machine learning for source code applied off-the-shelf machine learning tools with hand-extracted features. The second wave, reviewed here, avoids manual feature extraction and uses the source code itself within machine learning heavily drawing inspiration from existing machine learning methods in NLP and elsewhere. The third wave promises new machine learning models informed by programming language semantics. What form will it take?

At the time of this writing, machine learning, and deep learning in particular, is enjoying rock-star status among research fields. Despite its current (perhaps ephemeral) popularity, it is not a panacea. In some cases, a machine learning model may not be required (*e.g.* when the problem is deterministic) and, in other cases, simple models can outperform advanced, off-the-self deep learning methods, designed for non-code domains [65, 84]. Furthermore, over-engineering or under-engineering machine learning models usually leads to suboptimal results. Selecting a machine learning model for a specific problem necessitates questioning if a specific model is fit for the target application. Strong baseline methods are needed to estimate if a performance improvement justifies the added complexity of a model. In short, the right tools should be used for the right job, and machine learning is no exception.

Bridging Representations and Communities. Programming language research and practice use a well-defined and widely useful set of representations, usually in a symbolic form. In contrast, machine learning research customarily works with continuous representations. Bridging the gap between these representations by allowing machine learning systems to reason with programming language representations, is an important challenge. Handling unambiguous code has already led to a combination of probabilistic methods with "formal" constraints, that limit the probabilistic model to valid code. For example, Maddison and Tarlow [126] limit their model to generate syntactically correct and scope-respecting code while Raychev et al. [165] easily create a highly-structured model of a program taking advantage of its unambiguous form. Introducing better representations that bridge the gap between machine learning and source code will allow the probabilistic models of code to reason about the rich structure and semantics of code, without resorting to proxies. The core problem in this area is the lack of understanding of machine learning from the programming language community and vice-versa. At the same time, new machine learning methods that can handle programming language structures in its full complexity at scale need to be researched.

A major obstacle here is engineering systems that efficiently and effectively combine the probabilistic world of machine learning and the formal, logic-based world of code analysis. One approach, taken by several authors [9, 127, 165, 171], is to relax formal systems into probabilistic. Such systems, however, lack the guarantees formal systems (*e.g.* soundness) often provide. Alemi et al. [3], Balog et al. [20] and Loos et al. [123] follow a second approach that maintains soundness: they learn input and context-specific heuristics that efficiently guide search-based methods, such as theorem proving and program synthesis.

All of approaches to source code modeling must decide whether to explicitly model the structure and constraints of source code or to rely on general methods with adequate capacity. One one hand, using well-known, domain-generic machine learning methods has the advantage that the models are well-understood and rarely require significant effort or expertise to apply. On the other hand, designing models with built-in inductive biases for the problem domain usually performs better with less data, at the cost of manually designing and debugging domain (even problem-specific) networks. One such promising direction is modular neural network architectures. Such architectures decompose the network into components that are combined based on the problem instance. For source code models, such architectures can derive its structure through static analyses. These architectures have been useful for visual question answering in NLP. Andreas et al. [17] create neural networks by composing them from "neural modules", based on the input query structure. Similarly, we believe that such architectures will be useful within probabilistic models of source code. An early example is the work of Allamanis et al. [8] who design a neural network based on the output of data flow analysis. Such architectures should not only be effective for bridging representations among communities but — as we will discuss next — can combat issues with compositionality, sparsity and generalization. Nevertheless, issues that arise in static analyses, such as path explosion, will still need to be addressed.

Data Sparsity, Compositionality and Strong Generalization. The principle of reusability in software engineering creates a form of sparsity in the data, where it is rare to find multiple source code elements that perform exactly the same tasks. For example, it is rare to find hundreds of database systems, whereas one can easily find thousands of news articles on a popular piece of news. The exceptions, like programming competitions and student solutions to programming assignments, are quite different from industrial code. This suggests that there are many opportunities for researching machine learning models and inference methods that can handle and generalize from the highly-structured, sparse and composable nature of source code data. Do we believe in the unreasonable effectiveness of data [83]? Yes, but we do not have sufficient data.

Although code and text are both intrinsically extensible, code pushes the limit of existing machine learning methods in terms of representing composition. This is because most natural language methods rarely define novel, previously unseen, terms, with the possible exception of legal and scientific texts. In contrast, source code is inherently extensible, with developers constantly creating new terms (*e.g.* by defining new functions and classes) and combining them in still higher-level concepts. Compositionality refers to the idea that the meaning of some element can be understood by composing the meaning of its constituent parts. Recent work [86] has shown that deep learning architecture can learn some aspects of compositionality in text. Machine learning for highly compositional objects remains challenging, because it has proven hard to capture relations between objects, especially across abstraction levels. Such challenges arise even when considering simple code-like expressions [9]. However, if sufficient progress is to be made, representing source code artifacts in machine learning will improve significantly, positively affecting other downstream tasks. For example, learning composable models that can combine meaningful representations of variables into meaningful representations of expressions and functions will lead to much stronger generalization performance.

Data sparsity is still an important and unsolved problem. Although finding a reasonably large amount of source code is relatively easy, it is increasingly hard to retrieve some representations of source code. Indeed, it is infeasible even to compile all of the projects in a corpus of thousands of projects, because compiling a project requires understanding how the project handles external dependencies, which can sometimes be idiosyncratic. Furthermore, computing or acquiring semantic properties of existing, real-world code (*e.g.* purity of a function [60] or pre-/post- conditions [90])

is hard to do, especially at scale. Scalability also hampers harvesting run-time data from real-world programs: it is challenging to acquire substantial run-time data even for a single project. Exploring ways to synthesize or transform "natural" programs that perform the same task in different ways is a possible way ahead. Another promising direction to tackle this issue is by learning to extrapolate from run-time data (*e.g.* collected via instrumentation of a test-suite) to static properties of the code [4]. Although this is inherently a noisy process, achieving high accuracy may be sufficient, thanks to the inherent ability of machine learning to handle small amounts of noise.

Strong generalization also manifests as a deployability problem. Machine learning models, especially when they have become effective, are often so large that they are too large for a developer's machine, but using the cloud raises privacy[6] concerns and prevents offline coding. When under development and tooling is needed, code evolves quickly, subjecting models to constant concept drift and necessitating frequent retraining which can be extremely slow and costly. Addressing this deployability concern is an open problem and requires advances in machine learning areas such as transfer learning and one-shot learning. For example, say a program P uses libraries A and B, which have been shipped with the models M_A and M_B. Could we save time training a model for P by transferring knowledge from M_A and M_B?

Finally, source code representations are multifaceted. For example, the token-level "view" of source code is quite different from a data flow view of code. Learning to exploit multiple views simultaneously can help machine learning models generalize and tackle issues with data sparsity. Multi-view [193] and multi-modal learning (*e.g.* Gella et al. [70]), areas actively explored in machine learning, aim to achieve exactly this. By combining multiple representations of data, they aim to improve upon the performance on various tasks, learning to generalize using multiple input signals. We believe that this is a promising future direction that may allow us to combine probabilistic representations of code to achieve better generalization.

Measures. To train and evaluate machine learning models, we need to easily measure their performance. These measures allow the direct comparison of models and have already lead to improvements in multiple areas, such as code completion (Section 5.1). Nonetheless, these measures are imprecise. For instance, probabilistic recommender systems define a probability density over suggestions whose cross-entropy can be computed against the empirical distribution in test data. Although cross-entropy is correlated with suggestion accuracy and confidence, small improvements in cross entropy may not improve accuracy. Sometimes the imprecision is due to unrealistic use case assumptions. For example, the measures for LM-based code completion tend to assume that code is written sequentially, from the first token to the last one. However, developers rarely write code in such a simple and consistent way [158]. Context-based approaches assume that the available context (*e.g.* other object usages in the context) is abundant, which is not true in a real editing scenarios. Researchers reporting keystrokes saved have usually assumed that code completion suggestions are continuously presented to the user as she is typing. When the top suggestion is the target token, the user presses a single key (*e.g.* return) to complete the rest of the target.

Furthermore, some metrics that are widely used in NLP are not suited for source code. For example, BLEU score is not suitable for measuring the quality of output source code (*e.g.* in transducer models) because it fails to account for the fact that the syntax is known in all programming languages, so the BLEU score may be artificially "inflated" for predicting deterministic syntax. Second, the granularity over which BLEU is computed (*e.g.* per-statement *vs.* per-token) is controversial. Finally, syntactically diverse answers may be semantically equivalent, yielding a low BLEU score while

[6]https://www.theregister.co.uk/2017/07/25/kite_flies_into_a_fork/

being correct. Finding new widely-accepted measures for various tasks will allow the community to build reliable models with well-understood performance characteristics.

6.2 New Domains

Here, we visit a number of domains to which machine learning has not yet been systematically applied and yet suffer from uncertainty problems that machine learning is particularly well suited to address, promising new advances.

Debugging. Debugging is a common task for software engineers [191]. Debugging is like trying to find a needle in the haystack; a developer has to recognize relevant information from the deluge of available information. A multitude of tools exist in this area whose main goal is to visualize a program's state during its execution. Probabilistic models of source code could help developers, such as by filtering highly-improbable program states. Statistical debugging models, such as the work of Zheng et al. [198, 199] and Liblit et al. [117] are indicative of the possibilities within this area. Further adding learning within debugging models may allow further advances in statistical debugging. However, progress in this area is impeded by the combination of lack of data at a large scale and the inherent difficulty of pattern recognition in very high-dimensional spaces. Defects4J [100] — a curated corpus of bugs — could further prove useful within machine learning for fault prediction. Furthermore, collecting and filtering execution traces to aid debugging is another challenge for which machine learning is well-suited. Collection requires expensive instrumentation, which can introduce Heisenbugs, bugs masked by the overhead of the instrumentation added to localize them. Here the question is "Can machine learning identify probe points or reconstruct more complete traces from partial traces?" Concerning filtering traces, machine learning may be able to find interesting locations, like the root cause of bugs. Future methods should also be able to generalize across different programs, or even different revisions of the same program, a difficult task for existing machine learning methods.

Traceability. Traceability is the study of links among software engineering artifacts. Examples include links that connect code to its specification, the specification to requirements, and fixes to bug reports. Developers can exploit these links to better understand and maintain their code. Usually, these links must be recovered. Information retrieval has dominated link recovery. The work of Guo et al. [79] and Le et al. [113] suggests that learning better (semantic) representations of artifacts can successfully, automatically solve important traceability problems.

Two major obstacles impede progress: lack of data and a focus on generic text. Tracing discussions in email threads, online chat rooms (*e.g.* Slack), documents and source code would be extremely useful, but no publicly available and annotated data exists. Additionally, to date, NLP research has mostly focused on modeling generic text (*e.g.* from newspapers); technical text in conversational environments (*e.g.* chatbots) has only begun to be researched. StackOverflow presents one such interesting target. Although there are hundreds of studies that extract useful artifacts (*e.g.* documentation) from StackOverflow, NLP methods — such as dependency parsing, co-reference analysis and other linguistic phenomena — have not been explored.

Code Completion and Synthesis. Code completion and synthesis using machine learning are two heavily researched and interrelated areas. Despite this fact, to our knowledge, there has been no full scale comparison between LM-based [87, 144, 166] and structured prediction-based autocompletion models [33, 159]. Although both types of systems target the same task, the lack of a well-accepted benchmark, evaluation methodology and metrics has lead to the absence of a quantitative comparison that highlights the strengths and weaknesses of each approach. This highlights the necessity of widely accepted, high-quality benchmarks, shared tasks, and evaluation

metrics that can lead to comparable and measurable improvements to tasks of interest. NLP and computer vision follow such a paradigm with great success[7].

Omar et al. [149] discuss the challenges that arise from the fact that program editors usually deal with incomplete, partial programs. Although they discuss how formal semantics can extend to these cases, inherently any reasoning about partial code requires reasoning about the programmer's intent. Lu et al. [125] used information-retrieval methods for synthesizing code completions showing that simply retrieving snippets from "big code" can be useful when reasoning about code completion, even without a learnable probabilistic component. This suggests a fruitful area for probabilistic models of code that can assist editing tools when reasoning about incomplete code's semantics, by modeling how code could be completed.

Education. Software engineering education is one area that is already starting to be affected by this field. The work of Campbell et al. [35] and Bhatia and Singh [25] already provide an automated method for fixing syntax errors in student code, whereas Piech et al. [155], Wang et al. [182] suggest advancements towards giving richer feedback to students. Achieving reasonable automation can help provide high-quality computer science education to many more students than is feasible today. However, there are important challenges associated with this area. This includes the availability of highly-granular data where machine learning systems can be trained, difficulty with embedding semantic features of code into machine learning methods and the hardness of creating models that can generalize to multiple and new tasks. Student coursework submissions are potentially a ripe application area for machine learning, because here we have available many programs, from different students, which are meant to perform the same tasks. An especially large amount of such data is available in Massive Open Online Courses (MOOCs). This opens exciting possibilities, such as providing granular and detailed feedback, curriculum customization and other intelligent tutoring systems can significantly change computer science education.

Assistive Tools. Probabilistic models have allowed computer systems to handle noisy inputs such as speech, and handwritten text input. In the future, probabilistic models of source code may enable novel assistive IDEs, creating inclusive tools that improve upon conventional methods of developer-computer interaction and provide inclusive coding experiences.

7 RELATED RESEARCH AREAS

A variety of related research areas within software engineering and programming languages overlap with the area of statistical modeling of code. One of the most closely related research areas is *mining software repositories* (MSR) and *big code*. MSR is a well-established, vibrant and active field; the idea is to mine the large amounts of source code data and meta-data available in open-source (and commercial) repositories to gain valuable information, and use this information to enhance both tools and processes. The eponymous flagship conference is now in its 15^{th} iteration. "Big Code" is a synonymous neologism, created by DARPA's MUSE program[8] to borrow some branding shine from the well-marketed term *"Big Data"*. The MSR field's early successes date back to work by Zimmermann et al. [201], Williams and Hollingsworth [189], Gabel and Su [66] and Acharya et al. [1] on mining API protocols from source code bases. These approaches used pragmatic counting techniques, such as frequent item-set mining. Mining software repositories is in one sense a broader field than statistical models of code, as rather than focusing on code alone, MSR considers the full spectrum of software engineering data that can be derived from the software engineering

[7] See https://qz.com/1034972/ for a popular account of the effect of large-scale datasets in computer vision.
[8] http://science.dodlive.mil/2014/03/21/darpas-muse-mining-big-code/

process, such as process measures, requirement traceability, commit logs. Additionally, research in malware detection is related to probabilistic models of code [18, 36].

Another active area at the intersection machine learning and programming language research is *probabilistic programming* [73]. This might appear to be related to statistical models of code, but in fact there is a fundamental difference; essentially, probabilistic programming works in the reverse direction. Probabilistic programming seeks to deploy programming language concepts to make it easier for developers to write new machine learning algorithms. Statistical code models seek to apply machine learning concepts to make it easier for developers to write new programs. In some sense, the two areas are dual to each other. That being said, one can certainly imagine completing the cycle and attempting to develop statistical code models for probabilistic programming languages. This could be a fascinating endeavor as probabilistic programming grows in popularity to the extent that large corpora of probabilistic programs become available.

In software engineering, the term *modeling* often refers to formal specifications of program behavior, which are clearly a very different kinds of models than those described here. Combining formal models of semantics with statistical models of source code described in this review would be an interesting area for future research. There is also some work on probabilistic models of code that do *not* have a learning component, such as Liblit et al. [117]. Within machine learning, there has been an interesting recent line of work on neural abstract machines [75, 167, 168], which extend deterministic automata from computer science, such as pushdown automata and Turing machines, to represent differentiable functions, so that the functions can be estimated by techniques from machine learning. To the best of our knowledge, this intriguing line of work does not yet consider source code, unlike the work described in this review. Finally, semantic parsing is a vibrant research area of NLP that is closely related to the idea of program synthesis from natural language; see Section 5.5 for more discussion.

8 CONCLUSIONS

Probabilistic models of source code have exciting potential to support new tools in almost every area of program analysis and software engineering. We reviewed existing work in the area, presenting a taxonomy of probabilistic machine learning source code models and their applications. The reader may appreciate that most of the research contained in this review was conducted within the past few years, indicating a growth of interest in this area among the machine learning, programming languages and software engineering communities. Probabilistic models of source code raise the exciting opportunity of *learning* from existing code, probabilistically reasoning about new source code artifacts and transferring knowledge between developers and projects.

A SUPPLEMENTARY MATERIALS

REFERENCES

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[2] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using machine translation for converting Python 2 to Python 3 code.* Technical Report.

[3] Alex A Alemi, Francois Chollet, Geoffrey Irving, Christian Szegedy, and Josef Urban. 2016. DeepMath–Deep Sequence Models for Premise Selection. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*.

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. 2016. *Mining Semantic Loop Idioms from Big Code.* Technical Report. https://www.microsoft.com/en-us/research/publication/mining-semantic-loop-idioms-big-code/

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

Table 4. Some datasets, available online, used in research of Probabilistic Models of Source Code (sorted alphabetically). Links are clickable in the digital version.

Reference	Short Description	Link
Allamanis and Sutton [11]	Deduplicated snapshot of all Java GitHub projects with least one fork.	link
Allamanis et al. [10]	Parsed source code for 11 highly-ranked Java GitHub projects.	link
Barone and Sennrich [21]	Parallel corpus of 150k Python function declarations, docstrings and bodies.	link
Cerulo et al. [37]	Free text data with source code "islands".	link
Dyer et al. [54]	800k+ code repositories, software to support queries and mining	link
Iyer et al. [96]	Source code snippets with their StackOverflow title.	link
Kushman and Barzilay [110]	Dataset for generating regular expressions from natural language.	link
Lin et al. [119]	Text to Bash commands corpus.	link
Ling et al. [120]	Text descriptions and code for card game.	link
Raychev et al. [164]	A dataset of deduplicated JavaScript files and their ASTs extracted from GitHub.	link
Raychev et al. [164]	A dataset of deduplicated Python files and their ASTs extracted from GitHub.	link
Oda et al. [146]	Parallel corpus of Python code and pseudocode in English and Japanese.	link

[6] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[7] Miltiadis Allamanis and Marc Brockschmidt. 2017. SmartPaste: Learning to Adapt Source Code. *arXiv preprint arXiv:1705.07867* (2017).

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[9] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. 2017. Learning Continuous Semantic Representations of Symbolic Expressions. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[10] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[11] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*.

[12] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[13] Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[14] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. 2016. A study of Visual Studio usage in practice. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

[15] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*.

[16] Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. 2017. Neural Attribute Machines for Program Generation. *arXiv preprint arXiv:1705.09231* (2017).

[17] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Learning to Compose Neural Networks for Question Answering. In *Proceedings of NAACL-HLT*.

[18] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security Symposium*.

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[20] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[21] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[22] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2017. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. (2017).

[23] Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *arXiv preprint arXiv:1705.07962* (2017).

[24] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* (2010).

[25] Sahil Bhatia and Rishabh Singh. 2018. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[26] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. *arXiv preprint arXiv:1611.08307* (2016).

[27] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[28] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Programming with "Big Code": Lessons, Techniques and Applications. In *LIPIcs-Leibniz International Proceedings in Informatics*.

[29] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[30] David M Blei. 2012. Probabilistic Topic Models. *Commun. ACM* (2012).

[31] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *International Static Analysis Symposium*. Springer.

[32] Peter John Brown. 1979. *Software Portability: an advanced course*. CUP Archive.

[33] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[34] Raymond PL Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[35] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*.

[36] Lei Cen, Christoher S Gates, Luo Si, and Ninghui Li. 2015. A probabilistic discriminative model for android malware detection with decompiled source code. *IEEE Transactions on Dependable and Secure Computing* (2015).

[37] Luigi Cerulo, Massimiliano Di Penta, Alberto Bacchelli, Michele Ceccarelli, and Gerardo Canfora. 2015. Irish: A Hidden Markov Model to detect coded information islands in free text. *Science of Computer Programming* (2015).

[38] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)* (2017).

[39] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)* (2009).

[40] Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language* (1999).

[41] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. *Syntax, Semantics and Structure in Statistical Translation* (2014).

[42] Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*.

[43] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *Journal of Machine Learning Research (JMLR)* (2010).

[44] Christopher S Corley, Kostadin Damevski, and Nicholas A Kraft. 2015. Exploring the use of deep learning for feature location. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*.

[45] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE analyzer. In *ESPO*. Springer.

[46] William Croft. 2008. Evolutionary linguistics. *Annual Review of Anthropology* (2008).

[47] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017a. End-to-end Deep Learning of Optimization Heuristics. In *International Conference on Parallel Computing Technologies*.

[48] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017b. Synthesizing benchmarks for predictive modeling. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[49] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).

[50] Florian Deißenböck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* (2006).

[51] Yuntian Deng, Anssi Kanervisto, Jeffrey Ling, and Alexander M Rush. 2017. Image-to-Markup Generation with Coarse-to-Fine Attention. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[52] Premkumar Devanbu. 2015. New initiative: the naturalness of software. In *Proceedings of the International Conference*

on Software Engineering (ICSE).

[53] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[54] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[55] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. 2017. Learning to Infer Graphics Programs from Hand-Drawn Images. *arXiv preprint arXiv:1707.09627* (2017).

[56] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*.

[57] Michael D Ernst. 2017. Natural language is a programming language: Applying natural language processing to software development. In *LIPIcs-Leibniz International Proceedings in Informatics*.

[58] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R Brandt, and Michael S Bernstein. 2014. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the Annual ACM Conference on Human Factors in Computing Systems*.

[59] John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. 2017. Neural functional programming. 2016. *Proceedings of the International Conference on Learning Representations (ICLR)* (2017).

[60] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. 2008. Verifiable functional purity in java. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 161–174.

[61] Eclipse Foundation. Code Recommenders. www.eclipse.org/recommenders. (????). Visited June 2017.

[62] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltos Allamanis, Mirella Lapata, and Charles Sutton. 2017. Autofolding for Source Code Summarization. *IEEE Transactions on Software Engineering (TSE)* (2017).

[63] Jaroslav Fowkes and Charles Sutton. 2015. Parameter-Free Probabilistic API Mining at GitHub Scale. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[64] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[65] Wei Fu and Tim Menzies. 2017. Easy over Hard: A Case Study on Deep Learning. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[66] Mark Gabel and Zhendong Su. 2008. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[67] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[68] Rosalva E Gallardo-Valencia and Susan Elliott Sim. 2009. Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*.

[69] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428* (2016).

[70] Spandana Gella, Mirella Lapata, and Frank Keller. 2016. Unsupervised Visual Sense Disambiguation for Verbs using Multimodal Embeddings. In *Proceedings of NAACL-HLT*.

[71] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* (2015).

[72] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. www.deeplearningbook.org.

[73] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[74] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. 2012. Traceability fundamentals. In *Software and Systems Traceability*. Springer.

[75] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. *arXiv preprint arXiv:1410.5401* (2014).

[76] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[77] Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*.

[78] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, and others. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* (2017).

[79] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically enhanced software traceability using deep

learning techniques. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[80] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep Reinforcement Learning for Programming Language Correction. *arXiv preprint arXiv:1801.10467* (2018).

[81] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the Conference of Artificial Intelligence (AAAI)*.

[82] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

[83] Alon Halevy, Peter Norvig, and Fernando Pereira. 2009. The unreasonable effectiveness of data. *IEEE Intelligent Systems* (2009).

[84] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[85] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this?: Evaluating code contributions with language models. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*.

[86] Felix Hill, KyungHyun Cho, Anna Korhonen, and Yoshua Bengio. 2016. Learning to Understand Phrases by Embedding the Dictionary. *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)* (2016).

[87] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* (2016).

[88] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[89] Geoffrey E Hinton. 1984. Distributed representations. (1984).

[90] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. DOI: http://dx.doi.org/10.1145/363235.363259

[91] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* (1997).

[92] Reid Holmes, Robert J Walker, and Gail C Murphy. 2005. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes*.

[93] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. 2014. Using web corpus statistics for program analysis. In *ACM SIGPLAN Notices*.

[94] Xing Hu, Yuhan Wei, Ge Li, and Zhi Jin. 2017. CodeSum: Translate Program Language to Natural Language. *arXiv preprint arXiv:1708.01837* (2017).

[95] Andrew Hunt and David Thomas. 2000. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional.

[96] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[97] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[98] Daniel D Johnson. 2016. Learning graphical state transitions. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[99] Dan Jurafsky. 2000. *Speech & Language Processing* (3 ed.). Pearson Education.

[100] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

[101] Neel Kant. 2018. Recent Advances in Neural Program Synthesis. *arXiv preprint arXiv:1802.02353* (2018).

[102] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*.

[103] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. 2015. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078* (2015).

[104] Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*.

[105] Donald E Knuth. 1984. Literate programming. *Comput. J.* (1984).

[106] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam A Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.

[107] Rainer Koschke. 2007. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[108] Ted Kremenek, Andrew Y Ng, and Dawson R Engler. 2007. A Factor Graph Model for Software Bug Finding. In *Proceedings of the International Joint Conference on Artifical intelligence (IJCAI)*.

[109] Roland Kuhn and Renato De Mori. 1990. A cache-based natural language model for speech recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* (1990).

[110] Nate Kushman and Regina Barzilay. 2013. Using Semantic Unification to Generate Regular Expressions from Natural Language. In *Proceedings of Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

[111] Tessa Lau. 2001. *Programming by demonstration: a machine learning approach*. Ph.D. Dissertation. University of Washington.

[112] Quoc V Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[113] Tien-Duy B Le, Mario Linares-Vásquez, David Lo, and Denys Poshyvanyk. 2015. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *Proceedings of the International Conference on Program Comprehension (ICPC)*.

[114] Dor Levy and Lior Wolf. 2017. Learning to Align the Source Code to the Compiled Object Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[115] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated Graph Sequence Neural Networks. *Proceedings of the International Conference on Learning Representations (ICLR)* (2016).

[116] Percy Liang, Michael I Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[117] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*.

[118] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. 2017. *Program synthesis from natural language using recurrent neural networks*. Technical Report UW-CSE-17-03-01. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA.

[119] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *International Conference on Language Resources and Evaluation*.

[120] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomas Kocisky, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[121] Han Liu. 2016. Towards better program obfuscation: optimization via language models. In *Proceedings of the 38th International Conference on Software Engineering Companion*.

[122] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*.

[123] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 2017. Deep Network Guided Proof Search. In *LPAR*.

[124] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. *arXiv preprint arXiv:1704.04856* (2017).

[125] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2017. Data-Driven Program Completion. *arXiv preprint arXiv:1705.09042* (2017).

[126] Chris Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[127] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[128] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2013).

[129] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[130] Kim Mens and Angela Lozano. 2014. Source code-based recommendation systems. In *Recommendation Systems in Software Engineering*. Springer.

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

[132] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Conference of Artificial Intelligence (AAAI)*.

[133] Dana Movshovitz-Attias and William W. Cohen. 2013. Natural Language Models for Predicting Programming

Comments. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[134] Dana Movshovitz-Attias and William W. Cohen. 2015. KB-LDA: Jointly learning a knowledge base of hierarchy, relations, and facts. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[135] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017a. Bayesian Sketch Learning for Program Synthesis. *arXiv preprint arXiv:1703.05698* (2017).

[136] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017b. Finding Likely Errors with Bayesian Specifications. *arXiv preprint arXiv:1703.01370* (2017).

[137] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2015. Neural Programmer: Inducing latent programs with gradient descent. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[138] Graham Neubig. 2016. Survey of Methods to Generate Natural Language from Source Code. *http://www.languageandcode.org/nlse2015/neubig15nlse-survey.pdf* (2016).

[139] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[140] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[141] Anh T Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[142] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[143] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[144] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[145] Haoran Niu, Iman Keivanloo, and Ying Zou. 2016. Learning to rank code examples for code search engines. *Empirical Software Engineering (ESEM)* (2016).

[146] 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. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[147] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

[148] Cyrus Omar. 2013. Structured statistical syntax tree prediction. In *Proceedings of the Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*.

[149] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A Hammer. 2017. Toward Semantic Foundations for Program Editors. (2017).

[150] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[151] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[152] Terence Parr and Jurgen J. Vinju. 2016. Towards a universal code formatter through machine learning. In *SLE*.

[153] Jibesh Patra and Michael Pradel. 2016. Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data. (2016).

[154] Hung Viet Pham, Phong Minh Vu, Tung Thanh Nguyen, and others. 2016. Learning API usages from bytecode: a statistical approach. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[155] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[156] Matt Post and Daniel Gildea. 2009. Bayesian learning of a tree substitution grammar. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[157] Michael Pradel and Koushik Sen. 2017. Deep Learning to Find Bugs. (2017).

[158] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2016. Evaluating the evaluations of code recommender systems: a reality check. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[159] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2015).

[160] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. In *Proceedings of the Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*.

[161] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language To Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[162] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[163] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[164] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

[165] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

[166] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*.

[167] Scott Reed and Nando de Freitas. 2016. Neural Programmer-Interpreters. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[168] Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. 2017. Programming with a Differentiable Forth Interpreter. In *ICML*.

[169] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2010. Recommendation systems for software engineering. *Software, IEEE* (2010).

[170] Martin P Robillard, Walid Maalej, Robert J Walker, and Thomas Zimmermann. 2014. *Recommendation systems in software engineering*. Springer.

[171] Tim Rocktäschel and Sebastian Riedel. 2017. End-to-end Differentiable Proving. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*.

[172] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[173] Juliana Saraiva, Christian Bird, and Thomas Zimmermann. 2015. Products, developers, and milestones: how should I build my N-Gram language model. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[174] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[175] Abhishek Sharma, Yuan Tian, and David Lo. 2015. NIRMAL: Automatic identification of software relevant tweets leveraging language model. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

[176] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*.

[177] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*.

[178] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[179] Christoph Treude and Martin P Robillard. 2016. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[180] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[181] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[182] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Deep Knowledge Tracing On Programming Exercises. In *Conference on Learning @ Scale*.

[183] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016a. Bugram: bug detection with n-gram language models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*.

[184] Song Wang, Taiyue Liu, and Lin Tan. 2016b. Automatically learning semantic features for defect prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[185] Xin Wang, Chang Liu, Richard Shin, Joseph E. Gonzalez, and Dawn Song. 2016c. Neural Code Completion. (2016). https://openreview.net/pdf?id=rJbPBt9lg.

[186] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings*

of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE).

[187] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the International Conference on Automated Software Engineering (ASE).*

[188] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR).*

[189] Chadd C Williams and Jeffrey K Hollingsworth. 2005. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering (TSE)* (2005).

[190] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann.

[191] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)* (2016).

[192] Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the Working Conference on Mining Software Repositories (MSR).*

[193] Chang Xu, Dacheng Tao, and Chao Xu. 2013. A survey on multi-view learning. *arXiv preprint arXiv:1304.5634* (2013).

[194] Shir Yadid and Eran Yahav. 2016. Extracting code from programming tutorial videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.*

[195] Eran Yahav. 2015. Programming with "Big Code". In *Asian Symposium on Programming Languages and Systems.* Springer, 3–8.

[196] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)* (2017).

[197] Wojciech Zaremba and Ilya Sutskever. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

[198] Alice X Zheng, Michael I Jordan, Ben Liblit, and Alex Aiken. 2003. Statistical debugging of sampled programs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS).*

[199] Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the International Conference on Machine Learning (ICML).*

[200] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint arXiv:1709.00103* (2017).

[201] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering (TSE)* (2005).