Programming with "Big Code": Lessons, Techniques and Applications

Pavol Bielik, Veselin Raychev, and Martin Vechev

Department of Computer Science, ETH Zurich, Switzerland
<firstname>.<lastname>@inf.ethz.ch

Abstract

Programming tools based on probabilistic models of massive codebases (aka "Big Code") promise to solve important programming tasks that were difficult or practically infeasible to address before. However, building such tools requires solving a number of hard problems at the intersection of programming languages, program analysis and machine learning.

In this paper we summarize some of our experiences and insights obtained by developing several such probabilistic systems over the last few years (some of these systems are regularly used by thousands of developers worldwide). We hope these observations can provide a guideline for others attempting to create such systems.

We also present a prediction approach we find suitable as a starting point for building probabilistic tools, and discuss a practical framework implementing this approach, called Nice2Predict. We release the Nice2Predict framework publicly – the framework can be immediately used as a basis for developing new probabilistic tools. Finally, we present programming applications that we believe will benefit from probabilistic models and should be investigated further.

1998 ACM Subject Classification G.3 Probability and Statistics, I.2.2 Automatic Programming, D.2.4 Software/Program Verification

Keywords and phrases probabilistic tools, probabilistic inference and learning, program analysis, open-source software

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.41

1 Introduction

The rapid rise of open source code repositories such as GitHub [9], BitBucket [4], CodePlex [8] and others enables unprecedented access to a large corpus of high quality code. This corpus of code can be a valuable resource: much programmer effort and time has gone into developing, testing, debugging, annotating and fixing bugs in these programs.

A natural question then is: can we leverage and learn from this resource in order to create *new kinds of programming tools* that help developers accomplish tasks that were previously difficult or practically infeasible?

To address this challenge, a direction we have been exploring in the last couple of years is that of building probabilistic models of massive codebases, and then using these models as a basis for new kinds of probabilistic programming tools. For example, we developed statistical tools that are able to successfully and automatically complete API sequences [20], translate between programming languages [12], and de-obfuscate JavaScript programs [21].

Challenges. Ensuring that these tools work precisely on arbitrary general purpose programs (e.g. Android or JavaScript programs) requires solving several challenges, including selecting the right probabilistic model for the particular application at hand, finding the right program representation to be "compiled" into that model (and learned over the large corpus), as

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 41–50

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

well as the appropriate learning and inference algorithms. However, it is often the case that simply taking existing algorithms is not enough as they do not scale to the large available corpus that we can learn from, taking days or even weeks to finish. Therefore, when designing learning and inference algorithms an interesting trade-off arises where one needs to trade-off precision for scalability such that realistic codebases can be handled.

This work. This paper has four objectives:

- To present the observations and insights obtained from building a variety of probabilistic programming tools,
- To describe an approach based on structured prediction that can serve as a basis for creating a variety of new probabilistic programming tools,
- To present a practical framework based on this approach, called Nice2Predict, that can be used as a foundation of new probabilistic programming tools, and
- To discuss programming applications that can benefit from probabilistic models and should be investigated further.

2 Observations and Insights

We start by describing our observations and insights accumulated from developing several probabilistic programming tools: code completion [20], language translation [12], and code de-obfuscation [21]. We hope that these insights will be useful to others who attempt to create new tools.

2.1 Probabilistic Models vs. Clustering

A useful distinction to make when discussing techniques for learning from massive codebases is that of using probability distributions (models) vs. clustering-and-retrieval methods. Clustering is a natural and popular approach: we first group "similar" programs together (based on some criteria for similarity) and then given a developer query, look into the "nearest" cluster (or some other cluster) to see if we can retrieve a solution that already solves the problem. Examples of clustering approaches are [24, 5, 23, 16]. These tools enable the developer to phrase queries and discover if other code has already solved the problem. In our experience, a key issue with the cluster-and-retrieve approach is that it is usually unable to produce new code, that is, to predict programs not seen in the training data, yet are likely. This approach also makes it difficult to provide a meaningful probability assignment to a proposed solution.

On the other side, with probabilistic models, we typically first abstract (represent) the program by breaking it down into smaller pieces (e.g. extract features from the program) over which we then learn a probabilistic model. These features (program representation) are usually quite specific to the type of programming task being solved. The advantage of probabilistic models is that they can usually provide predictions/solutions that are *not* found in the training data and can also associate probabilities with the proposed solution.

A key attribute of using probabilistic models is the presence of regularization or backoff. Regularization bounds the weights of the features to avoid creating a too complex model that only captures the properties of the training data – a problem also knows as over-fitting. A key step in achieving the precision of the JSNice code de-obfuscation [21] was to use cross-validation and pick an efficient regularization. Similarly, in our code completion system [20], we rely on a backoff technique [25] that was developed for n-gram statistical language models.

Backoff avoids over-fitting by considering only features with sufficient support in the training data and falling back to a simpler model for features with not enough support.

2.2 Probabilistic Models and Program Representations

Perhaps the most fundamental question when trying to build probabilistic models of code is deciding on the right program representation (features). What representation should be used when trying to learn a model over multiple programs? Representations that are too fine-grained and program specific or too coarse-grained and unable to differentiate between programs will not result in interesting commonalities being learned across many programs.

The program representation used is also connected with the particular model. For example, if one uses statistical language models (e.g. n-gram) [20], it may be natural to use a representation which consists of sequences of API sequences, however, it may be difficult to specify other features that are not sequence related (e.g. that the same variable is passed to multiple API invocations). Of course, it can also be the case that there are multiple probabilistic models which work with the same representation. For example, [20] uses both n-gram models and recurrent neural networks (RNNs).

Fundamentally, we believe that down the line, an important research item is finding the right probabilistic models and representations for programs. For instance, in the context of graphical models, popular models frequently used in computer vision are the Potts and the Ising models – images are usually converted to this representation and a subsequent prediction is then performed on that representation. Such models have very well studied theoretical properties and are equipped with suitable inference algorithms with provable guarantees. Can we find the corresponding fundamental representation for programs?

2.3 The Need for Clients

Developing probabilistic models of code is only part of the challenge. Even if one successfully builds such a model, it is not at all clear that this model will perform well for solving a particular programming challenge and will be able to handle realistic programs accurately. It is therefore critical to evaluate the model in the context of a particular client (e.g. code completion, etc.). In our experience, it is often the case that using a model by a particular client usually results in need to augment and fine-tune the model (e.g. by adding more features, etc.). For instance, in [12] we found out that the traditional score for measuring the quality of the learned model (the BLEU score) was not at all indicative of how well the model performs when translating new programs.

An interesting challenge here arises when one has to fine-tune the model. For instance, in the n-gram model, one needs to decide whether to use the 2-gram, the 3-gram, the particular smoothing method, etc. In natural language processing, there are key works which study these parameters and suggest "good" values [6] for certain models (e.g. the n-gram model).

Natural questions include: can we automatically come up with the model parameters based on the particular client? Are there appropriate metrics for the quality of the probabilistic model of code where the client is unknown, or do we always need to know what the client is?

2.4 The Need for Semantics

In our experience, it is important to leverage programming languages techniques (e.g. semantic program analysis) in order to extract quality information from programs upon which we learn the probabilistic model. That is, it is not advisable to learn directly over the program

syntax as done in various prior works [11, 2]. We believe that it is very difficult to design useful programming tools without any semantic code analysis.

In particular, unlike the domain of images (e.g. image recognition, etc.), programs are not data (e.g. a collection of pixels), but data transformers: they generate new data and program analysis techniques enable us to get access to this data.

For example, in the case of API code completion, if instead of API sequences, one generates any syntactic sequence, then the sequence will contain a lot of "noise" leading to much sparsity in the probabilistic model and relating tokens that are unrelated to the API prediction. However, statically extracting API sequences precisely is a challenging program analysis problem requiring alias and typestate analysis. Our experience is that accurate static analysis is important for extracting quality semantic information. Note that the extracted information does not need to be an over-approximation to be useful: it is possible that the extracted information is both and under- and an over- approximation (i.e. as in symbolic execution).

Similar situation arises in statistical programming language translation in [12] where we had to incorporate knowledge about the language grammar in the translation process in order to avoid the statistical translation generating grammatically incorrect programs. Here too, more semantic information would have been useful in preserving semantic equivalence between the source and the translated program.

3 Prediction Approach

In this section we provide an overview of the proposed approach for building probabilistic tools. The main benefit of this approach is that it supports structured prediction which, instead of predicting labels in isolation, uses the provided context into account to make a joint prediction across all labels. Binary or multi-class predictors (that predict features in isolation) are a special case of this model.

In particular, we use Conditional Random Fields (CRFs) which is a discriminative log-linear classifier for structured prediction. The CRFs is a relatively new model introduced by [15] and since have been used in various domains such as natural language processing, information retrieval, computer vision and gene prediction [18, 17, 3, 10, 15]. We believe that future research will leverage and extend these ideas from CRFs and apply them in the context of programs.

3.1 Conditional Random Fields (CRFs)

For each program (i.e. a prediction instance), we consider two vectors of properties $\mathbf{x}$ and $\mathbf{y}$ (we can think of properties broadly as program elements: types, code, method names, etc.). The vector $\mathbf{x}$ includes known properties (e.g. properties which can be computed from the input with existing program analysis techniques). The vector $\mathbf{y}$ captures inferred properties for the given (potentially partial) program. Each known property has value ranges in X while unknown properties range over Y, thus $\mathbf{x} \in X^*$ and $\mathbf{y} \in Y^*$. A CRF model has the following form:

$$P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z(\mathbf{x})} \exp(score(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda}))$$

where $Z(\mathbf{x})$ is an instance-specific partition function (also referred to as a normalization function), $\boldsymbol{\lambda}$ are weights learned in the training phase, and $score(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda})$ is a function that calculates a real valued score for given assignment $\mathbf{y}$ and $\mathbf{x}$. The partition function ensures

that $P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\lambda})$ is a valid probability distribution (i.e., sums to one) over the range of unknown properties Y and is defined as follows:

$$Z(\mathbf{x}) = \sum_{\mathbf{y} \in Y^*} \exp(score(\mathbf{y}, \mathbf{x}))$$

The scoring function $score(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda})$ assigns higher score to more likely assignments of $\mathbf{x}$ and $\mathbf{y}$ and is defined as follows:

$$score(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda}) = \sum_{i=1}^{k} \lambda_i f_i(\mathbf{y}, \mathbf{x}) = \boldsymbol{\lambda}^T \mathbf{f}(\mathbf{y}, \mathbf{x})$$

where f_i is i-th feature function $f : X^* \times Y^* \to \mathbb{R}$ with associated weight λ_i. For convenience we use vector notation $\boldsymbol{f}(x, y) = [f(x, y)_1 \ \ldots \ f(x, y)_k]$ for feature vector of k feature functions and $\boldsymbol{\lambda} = [\lambda_1 \ \ldots \ \lambda_k]$ for learned weights. The weights $\boldsymbol{\lambda}$ are learned from the training data.

Challenges. Conditional random fields define a probability distribution that is linear with respect to the learned weights $\boldsymbol{\lambda}$, but computing the actual probability with the formula $P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z(\mathbf{x})} \exp(\boldsymbol{\lambda}^T \mathbf{f}(\mathbf{y}, \mathbf{x}))$ requires evaluating an expensive $Z(\mathbf{x})$ function that computes a sum over all possible assignments of labels (program properties $\mathbf{y} \in Y^*$). To avoid this limitation, we focus on applications where we do not need to report the actual probability of the prediction, but only the ability to compare probabilities of different assignments of the properties $\mathbf{y} \in Y^*$.

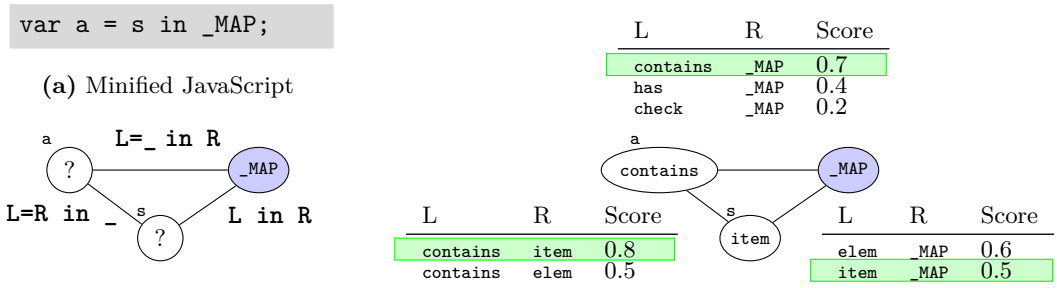
MAP Inference Query. The most important query in this context is that of MAP Inference (also known as Maximum APosteriori Inference):

$$\mathbf{y} = \operatorname*{argmax}_{\mathbf{y}' \in \mathcal{P}(Y)} P(\mathbf{y}' \mid \mathbf{x})$$

To answer this query, we do not need to evaluate the partition function since it is constant with respect to $\mathbf{y}'$ and thus $\mathbf{y} = \operatorname{argmax}_{\mathbf{y}' \in \mathcal{P}(Y)} \boldsymbol{\lambda}^T \mathbf{f}(\mathbf{y}', \mathbf{x})$. This MAP inference query requires solving an optimization problem – finding the labels that score the best. Typically, solving this problem requires leveraging properties of the feature functions $\mathbf{f}$.

Unfortunately, selecting an appropriate interference algorithm is challenging as: i) the exact inference problem is generally NP-hard and too expensive to be used in practice, and ii) even common approximate inference algorithms such as (loopy) belief propagation have shown to be at least an order of magnitude slower than greedy algorithms. Given the large-scale nature of the applications, we provide a greedy inference algorithm specifically designed to efficiently handle a massive range of the domain Y, implementation details of which can be found in [21].

Alternatives. The most important alternative discussed in literature avoids MAP inference altogether and replaces it with marginal inference. In marginal inference, we simply select for each property an assignment which maximizes its marginal probability. Such assignment however ignores dependencies between properties as it maximizes only the local probability of assignment to a single property. In contrast, optimizing the joint probability of the assignments to all the properties can lead to significant increase in accuracy as illustrated in the 9.3% improvement in the name prediction task [21].



(a) Minified JavaScript

(b) Dependency network

(c) Result of MAP inference

Figure 1 a) Code snippet of minified JavaScript, b) Dependency network build using properties and features extracted from a). The known properties are shown in blue (global variable `_MAP`) and unknown properties in white (variables `a` and `s`). c) An overview of the name inference procedure.

Naturally, there is an opportunity that an alternate algorithm will provide a meaningful approximation of the exact MAP inference. For example, one could first optimize all the labels in isolation and only then optimize by taking structural features into account. However, this should be done by taking the constraints of the problem into account.

Constraints. An important extension of the greedy MAP inference is to support hard constraints on the range of possible values y. For example, in a name prediction task, one constraint is that only non-duplicate names in scope are allowed. Such constraints cannot be neglected as they guarantee that the resulting MAP assignment is a feasible program. It is worth noting that common approximate inference algorithms do not support constraints over the range of possible assignments and therefore can produce non-feasible solutions. Integrating constraints into existing algorithms is an open problem which will need to be addressed as soon as the need for more accurate inference algorithms arises.

Example. To illustrate MAP inference, we predict variable names of the code snippet in Fig. 1 a) using JSNice [21] – a system for predicting variable names and types. We start by identifying two unknown properties corresponding to variables `a` and `b` together with one known property corresponding to global variable `_MAP`. Next, the program analysis extracts three feature functions which relate the properties pairwise. An example of feature function is `L in R` between properties `s` and `_MAP`, which captures the fact that they are used together as a left-hand and right-hand side of binary operator `in`. The properties and feature functions are succinctly represented as a dependency network Fig. 1 b), which is used as a graphical representation of CRF.

Given the dependency network, we are interested in performing the MAP inference. That is, we want to find an assignment to unknown properties such that it maximizes the scoring functions learned from the data. An illustration of MAP inference is shown in Fig. 1 c). The particular assignment of properties in this case demonstrates that the MAP inference computes a global maximum of the scoring functions, which does not necessarily correspond to the best local per property assignments (i.e., for properties `s` and `_MAP` only a second best scoring assignment was selected).

Extensions. The choice of features enable fast approximate MAP inference algorithm, but it has an important implication on the system capabilities. In particular, our current implementation of the system has the following limitations:

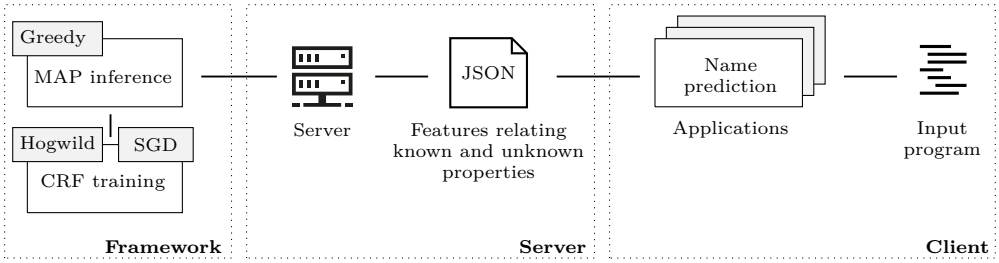


Figure 2 Overview of the framework architecture.

- Fixed domain of unknown properties. That is, only label values seen in the training data can be predicted.
- Pairwise feature functions. The feature functions can be defined only for pairs of properties and dependencies of higher cardinality are split into several pairwise feature functions.

In the context of previously discussed application of variable name prediction the above limitations are not critical to the system performance. However, they ultimately restrict the system's capabilities to predict arbitrary program properties.

4 Nice2Predict: A Prediction Framework

We believe an important part of the research on probabilistic programming tools is having an effective systems that works in practice and is shown to be effective when trained on real problems and code. Towards this, we discuss an implementation of the approach discussed above in scalable, efficient and extensible to new applications implementation available at:

`https://github.com/eth-srl/Nice2Predict`

In particular, the implementation is decoupled from the particular programming language (and from particular clients such as JSNice).

The architecture of Nice2Predict is shown in Fig. 2. The core library contains an efficient C++ implementation of both greedy MAP inference and state-of-the-art max-margin CRF training [21] which are both subject to future extensions. The CRF training currently supports two modes: parallel stochastic gradient descent (SGD) [26] and Hogwild [22] training. Further, the training is highly customizable by providing control over several important parameters such as learning rate, SVM margin and regularization. Additionally, we provide automatic cross-validation procedure which finds the best values for all parameters.

The server provides interoperation between the C++ core implementation and clients using different programming languages. Our communication format is JSON which most languages can easily write to and we provide a binding with a remote procedure call (RPC) over the HTTP server.

We chose to implement the inference engine as an external service to cover a range of clients. The clients provide the remaining steps: i) defining known and unknown properties, ii) defining features that relate properties, and iii) obtaining training data. That is, the client is mainly responsible for transforming an input program into a suitable representation which is then encoded to a JSON and sent as a query to the server.

Example client: predicting JavaScript names

We now briefly describe a client of our framework – a JavaScript deminifier implementation called UnuglifyJS which extracts properties and features as described in [21]. This client can be used as a starting point to help researchers and developers interested in using the Nice2Predict framework. The open-source implementation is available at:

`https://github.com/eth-srl/UnuglifyJS`

As discussed above, the client is responsible for three items: i) defining known and unknown properties, ii) defining features, and iii) obtaining training data. We start by describing the known and unknown properties. The known properties are program constants, objects properties, methods and global variables – that is, program parts which cannot be (soundly) renamed (e.g. the DOM APIs). The unknown properties are all local variables.

To define features, we first represent the properties in a graph where the nodes are properties, and each edge represents a relationship between them. We associate a label with each node - the value in the corresponding component from the vectors $\mathbf{x}$ and $\mathbf{y}$. Then, we define the feature functions to be indicator function for all possible labels connected by an edge. In the case of names, we introduce a feature function for all possible triples: $(z_1, z_2, type)$ where z_1 and z_2 are labels ($z_1, z_2 \in X \cup Y$) and *type* describes the type of the relation encoded in the graph edge. Finally, specifically for JavaScript names, we create a simple filter that detects minified code and performs training on non-minified code with meaningful variable names. Based on these names, the system learns the feature functions used for code deminification.

Further, for this prediction application, we provide an interactive walk-through available at `http://nice2predict.org/`. Once the user provides JavaScript, the resulting CRF graph is visualized and a Nice2Predict server performs MAP inference.

5 Applications

We next discuss several applications that we believe would benefit from and are worth investigating in the context of probabilistic models.

- *Statistical program synthesis*: in the last few years, we have seen renewed interest in various forms of program synthesis. Usually, these techniques are applied on a per-program basis (e.g. synthesize the program from a partial specification such as assertions, input-output examples, etc.). A natural direction is to consider program synthesis based on probabilistic models. We already made first steps in prior work [20] (in the context of code completion), but much more investigation is needed to understand the limits of statistical code synthesis.
- *Statistical invariant prediction*: an interesting direction is whether we can predict properties of programs by learning models of these properties over other programs already annotated with their properties. First steps in that direction were made in [21], where we predict simple types for JavaScript (checked by the Google Closure Compiler [7]) – can this approach be applied to infer useful contracts or ownership annotations in the style of Kremenek et al. [14]? Such a system could potentially complement existing static analysis tools and make it easier to annotate programs.
- *Statistical code beautification and refactoring*: a promising direction is leveraging probabilistic models for learning natural coding and naming conventions [21, 1]. Suggesting better names and code refactorings based on probabilistic models is useful in various

settings include software engineering (better code readability) and de-obfuscation as in JSNice[1].

- *Statistical error prediction*: a useful direction we have been exploring recently is discovering pieces of program code that are unlikely and suggesting alternative code that is more likely. This has applications in regular programming but also in the context of massive open online courses (MOOCs). In the context of concurrency, can we use statistical models to predict likely thread interference and suggest likely synchronization repairs?
- *Statistical language translation*: recently, we explored the direction of using statistical models to translate programming languages [12]. An interesting observation here is that statistical translation is sometimes able to learn how to translate entire patterns of code (i.e. how to map one set of APIs to another). One challenge here is obtaining a parallel corpus of translated programs – a possible direction is using Amazon Turk for obtaining such corpus.
- *Statistical code exploration*: an interesting application of statistical models is to guide the program exploration or test case generations to exercise specific parts (e.g. suspicious) of the codebase first. Such approach is particularly interesting for large scale projects which ensure their security via a fuzzing infrastructure – an automated the test case generation and crash detection.

Thinking of applications: from Computer vision to Programs. A useful place to think about applications is to consider the analogous problems in computer vision. For example, code deobfuscation can be seen as image denoisification, where noisy pixels correspond to obfuscated program parts. Code completion corresponds to image completion and code documentation is akin to words describing an image. This view is important not only for inspiring new applications, but by formalizing these problems in similar terms as in vision, we can transfer advances such as efficient feature learning [19] and a rich arsenal of inference techniques [13] to programming language problems.

6 Conclusion

In this paper, we presented four items: insights for building probabilistic programming tools, a prediction approach that can serve as a starting point for developing such tools, a practical framework based on this approach, and a set of applications worth exploring further.

References

1 Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Learning natural coding conventions. In *Proc. of the 22nd ACM SIGSOFT Int' Symp. on Foundations of Software Engineering (FSE'14), 2014*, pages 281–293, 2014.

2 Miltos Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *MSR*, 2013.

3 Julian Besag. On the Statistical Analysis of Dirty Pictures. *Journal of the Royal Statistical Society. Series B (Methodol.)*, 48(3):259–302, 1986.

4 Atlassian bitbucket. `https://bitbucket.org/`.

5 Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (ESEC/FSE'09)*, pages 213–222, 2009.

[1] `http://jsnice.org`

6 Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proc. of the 34th Annual Meeting on Association for Computational Linguistics (ACL'96)*, pages 310–318, 1996.

7 Google closure compiler. https://developers.google.com/closure/compiler/.

8 Atlassian bitbucket. `https://www.codeplex.com/`.

9 Github. `https://github.com/`.

10 Xuming He, Richard S. Zemel, and Miguel Á. Carreira-Perpiñán. Multiscale conditional random fields for image labeling. In *Proc. of the 2004 IEEE Conf. on Computer Vision and Pattern Recognition*, CVPR'04, 2004.

11 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE 2012*, 2012.

12 Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proc. of the 2014 ACM Int'l Symp.on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'14)*. ACM, 2014.

13 Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

14 Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI'06)*, OSDI'06, pages 161–176, Berkeley, CA, USA, 2006. USENIX Association.

15 John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of the 18th Int'l Conf. on Machine Learning (ICML'01)*, ICML'01, pages 282–289, 2001.

16 Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA'12*, 2012.

17 David Pinto, Andrew McCallum, Xing Wei, and W. Bruce Croft. Table extraction using conditional random fields. In *Proc. of the 26th Annual Int'l ACM SIGIR Conf. on Research and Development in Informaion Retrieval (SIGIR'03)*, pages 235–242, 2003.

18 Ariadna Quattoni, Michael Collins, and Trevor Darrell. Conditional random fields for object recognition. In *NIPS*, pages 1097–1104, 2004.

19 Nathan D. Ratliff, J. Andrew Bagnell, and Martin Zinkevich. (approximate) subgradient methods for structured prediction. In *AISTATS*, pages 380–387, 2007.

20 Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'14)*, pages 419–428. ACM, 2014.

21 Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting Program Properties from "Big Code". In *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'15)*, pages 111–124. ACM, 2015.

22 Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. of Neural Information Processing Systems Conf. (NIPS'11)*, pages 693–701, 2011.

23 Steven P. Reiss. Semantics-based code search. In *ICSE'09*, 2009.

24 Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE'07*, 2007.

25 Ian H. Witten and Timothy C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.

26 Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.