Concept-aware Live Programming

Integrating Topic Models for Program Comprehension into Live Programming Environments

Toni Mattis
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

CCS CONCEPTS

• **Computing methodologies** → *Mixture models*; • **Software and its engineering** → Integrated and visual development environments; **Object oriented languages**; Software reverse engineering;

KEYWORDS

Program comprehension, Concept assignment problem, Live programming, Topic models

ACM Reference format:
Toni Mattis. 2017. Concept-aware Live Programming. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 2 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079369

1 MOTIVATION

Live programing environments, such as Smalltalk, are valued for allowing program changes at run-time while the full execution state is accessible for inspection [12]. The resulting immediacy of feedback motivates continuous and fine-grained change, exploration, and checking of hypotheses the programmer might have generated by recognizing familiar names and structures.

However, liveness alone does not guarantee a high-level understanding of the system at hand. Programmers might be oblivious to the original mental model from which terminology and architecture were drawn [3, 8]. This promotes inconsistent naming, missing relevant dependencies during a change, duplicating functionality, or failure to find a variation point to add a new requirement.

A recent approach to allocate concepts to program parts are latent topic models, which identify concept-specific terminology chosen by programmers.

In the context of this work, we extend topic modeling to handle some of the information needs in a live programming environment. We hypothesize that machine learning and live programming can jointly contribute to program comprehension at the level of static code, run-time behavior, and state.

2 BACKGROUND AND RELATED WORK

Topic Models. A family of techniques to recover concepts from documents are probabilistic *topic models*, such as Latent Dirichlet Allocation (LDA) [5] or Hierarchical Dirichlet Processes (hDP) [13].

Programming '17, Brussels, Belgium
 978-1-4503-4836-2/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079369

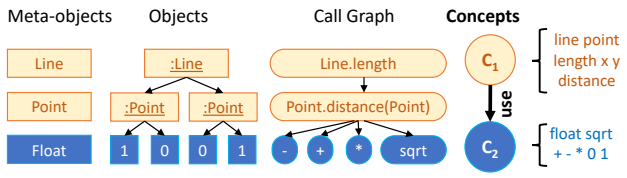

Figure 1: Simplified concept model simultaneously explaining code, state, and behavior (left) by grouping concept-specific terminology (right).

Topic models group words from a set of documents into a small number of *topics* based on frequent co-occurrence, and assign each document the proportions of how concerned it is with each topic, yielding a lossy compression of each document's word histogram.

Topics in Source Code. Recent work indicates that LDA recovers coarse-grained concepts from code reasonably well [2, 4, 7, 11]. Related models like hDP [13] for *hierarchies* of topics, or Stochastic Block Models [1] for *graph-structured* data remain understudied in this context. Nevertheless, they are of interest to our research due to their explanatory power. Graph- and trace-based code similarity measures suited for clustering are used in the field of aspect mining [9, 14]. Other approaches incorporate natural language to capture the meaning of code in terms of the vocabulary used by programmers to document its functionality [10, 15].

3 APPROACH

We consider a *concept* as a part of the real-world or technical domain of a program that can be distinguished from other concepts by a unique terminology (e.g. "line" and "point" as parts of the "geometry" concept, see Figure 1). Concepts are composed of other concepts by means of *abstraction*.

We require a *concept model* capable of answering the following questions about any given meta-object (e.g. method, class, stack frame) or object (e.g. instance) at run-time:

(1) Given a (meta-)object,
  (a) which concepts does it represent and in which role does it appear in the respective concept?
  (b) which concepts does it use for its implementation?
(2) Given a concept,
  (a) which (meta-)objects realize it?
  (b) which other concepts use it in their implementation?
  (c) which other concepts are used for its implementation?

We inferred these information needs from an initial case study involving the review of students projects, and the observation which

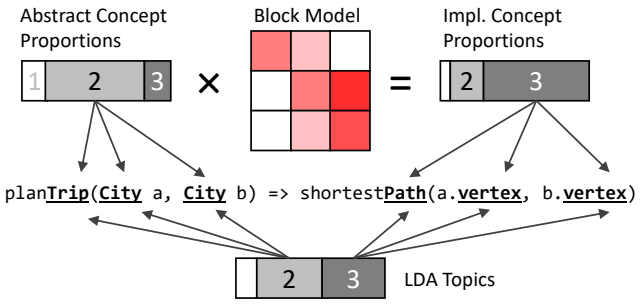

Figure 2: Example function definition under our concept model (upper part) and LDA (lower part).

questions are frequently asked when the live programming environment used throughout this course is explored during the semester.

Concept Model. We investigate a novel variation of LDA that incorporates a topic × topic matrix analogously to a *stochastic block model* learning which abstractions make use of which implementations (Figure 2). This model is capable of explaining the abstract vocabulary used in a public interface as topic mixture and regards identifiers chosen in the implementation as a different topic mixture derived from the abstract topics translated through the block model. It also serves as an edge predictor in the call graph and in the object graph, thereby offering the possibility to be further trained from run-time data in addition to static code. We implemented inference via Gibbs sampling similar to LDA [6].

Usage Inside the Environment. Besides visualizing and navigating the extracted model to get an initial system overview, concepts can be represented by color coding in the debugger's *call stack* or *object inspector*. *Syntax completion* can scope its proposals to concepts currently surrounding the completion site and offer the possibility to switch between lower and higher abstraction levels. The *code browser* and *editor* can recommend live objects rather than just related code artifacts. Code not aligned with the surrounding concepts yields *refactoring hints*.

4 OUTLOOK AND CONCLUSION

Our research is in an early stage and focused on the theoretical combination of lexical features with the graph structure of code, state, and behavior. The model presented here will likely change in the future. So far, we integrated data specific to an object-oriented live programming environment into a single probabilistic model and are currently working on effective integration into the standard tools of a Smalltalk environment. We plan an evaluation based on standard language modeling metrics, such as perplexity, as well as an assessment of its contribution to program comprehension in a Smalltalk environment after tool integration has progressed to a sufficient degree.

REFERENCES

[1] E. M. Airoldi, D. M. Blei, S. E. Fienberg, and E. P. Xing. Mixed Membership Stochastic Blockmodels. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 33–40. Curran Associates, Inc., 2009.
[2] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software Traceability with Topic Modeling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 95–104, New York, NY, USA, 2010. ACM.
[3] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of 1993 15th International Conference on Software Engineering*, pages 482–498, May 1993.
[4] D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt. Understanding LDA in Source Code Analysis. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 26–36, New York, NY, USA, 2014. ACM.
[5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.
[6] T. Griffiths. Gibbs sampling in the generative model of Latent Dirichlet Allocation, 2011.
[7] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining Concepts from Code with Probabilistic Topic Models. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 461–464, New York, NY, USA, 2007. ACM.
[8] P. Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253–261, May 1985.
[9] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen. Aspect Recommendation for Evolving Software. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 361–370, New York, NY, USA, 2011. ACM.
[10] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, Nov. 2015.
[11] A. M. Saeidi, J. Hage, R. Khadka, and S. Jansen. ITMViz: Interactive Topic Modeling for Source Code Analysis. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 295–298, Piscataway, NJ, USA, 2015. IEEE Press.
[12] S. L. Tanimoto. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press.
[13] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101, 2004.
[14] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering, 2004. Proceedings.*, pages 112–121, Nov. 2004.
[15] M. Zilberstein and E. Yahav. Leveraging a Corpus of Natural Language Descriptions for Program Similarity. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 197–211, New York, NY, USA, 2016. ACM.