Interactive systems for discovering population-level structure in code & data

Elena L. Glassman

December 12, 2017

I am a researcher in human-computer interaction working on programming systems. I design, build and evaluate interactive systems for population-level reasoning about large code and other data corpora. These systems support the discovery of emergent structure, as well as data-driven human and machine teaching to augment human intelligence.

Data-driven decisions are becoming increasingly ubiquitous. I work on the class of problems in which people must answer questions that require a population-level characterization of relevant data. For example, when programmers want to use an unfamiliar class in the Java API, the handful of examples they find in online documentation and Q&A forums may not answer their questions about the various ways that the Java class is used in production code in the wild or at their company. In massive programming and engineering courses, lecturers have to reason about what their students are learning; evidence to answer that question can be found in the landscape of code and designs their students submit. Data scientists are, by nature and necessity, concerned with the location of corner cases buried in their data before writing any code to process or manipulate that data. In my doctoral and postdoctoral work at MIT and UC Berkeley, I have created and deployed systems that make it possible to express and answer some of these important population-level questions in novel ways. I refer to this as **code and data demography**.

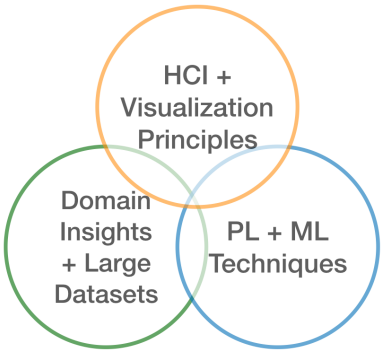

Figure 1: Common system components.

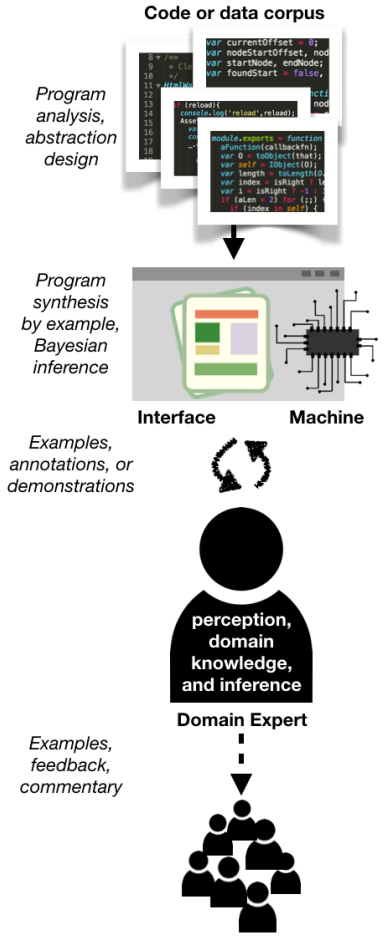

Figure 2: Common system architecture.

I HAVE FOCUSED ON large code corpora mined from open-source repositories on Github and collected from massive electrical engineering and computer science classes (Fig. 1, 2). I use **program analysis and visualization principles** to render large amounts of code in a way that works with human perceptual strengths. I incorporate interactive inference techniques from **program synthesis and machine learning** to help users discover structure and abstractions that can be automatically applied to new code and data. As a fellow at the Berkeley Institute of Data Science, I am excited to generalize this work to new types of data and applications to help data scientists, social scientists, and end-users.

To design systems that augment humans' innate ability to make structural inferences from data, I leverage theories from **psychology, the learning sciences, and cognitive science**. While machines can easily perform computation on millions of data points, humans can only consider a relatively small number of examples. And yet, humans have within them powerful engines for inference and abstraction, as well as a wealth of domain knowledge. Theories from educational psychology and cognitive science, like Variation Theory, [1] characterize

[1] M. Ling Lo. *Variation theory and the improvement of teaching and learning*. Göteborg: Acta Universitatis Gothoburgensis, 2012

how humans are likely to make generalizations from sets of examples that vary in both superficial and critical ways. Through user-centered design and automated inference, my systems illuminate sets of examples for domain experts to interact with and refine, in order to induce accurate representations of large datasets in the expert's mind. This example-based view of a corpus can also help experts pick and annotate better examples to teach concepts and abstractions to other people and machines.

Teachers and students as domain experts

Enrollment in introductory programming and data science courses is skyrocketing. Students have a shared semantic programming goal but they collectively create a large distribution of implementations. It is difficult for teachers to understand what is happening in their classes, and prohibitively labor-intensive to provide expert-level feedback at scale as often as it was possible before enrollments rose dramatically. It is both a challenge and an opportunity to reinvent how we teach students and how students teach each other.

With collaborators and mentees, I built, user-tested, and deployed five systems at MIT and UC Berkeley that enable teachers to see the distribution of student code submissions and misconceptions, discover unusually clever and poor choices, curate sets of examples that clarify important concepts, and give both class-level and automatically personalized individual feedback that can be reused in subsequent semesters. OVERCODE [2] (Fig. 3) is now integrated into UC Berkeley's largest on-campus programming course, CS61a, which regularly enrolls over 1500 students. Two students I mentor are now preparing to make OVERCODE available to all introductory Python programming teachers at any school or university this spring.

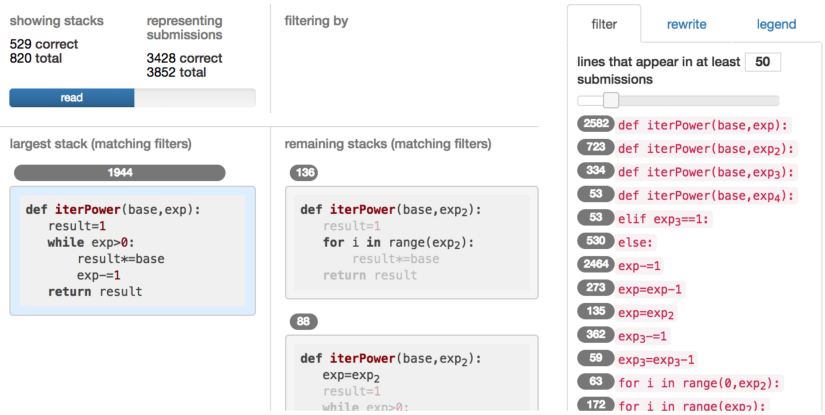

Figure 3: OVERCODE is an example of code demography for teachers in massive programming classes who want to understand the contents of the functions their students wrote.

OVERCODE is an example of code demography for visualizing and exploring thousands of solutions to the same programming problem. It uses both static and dynamic analysis to cluster similar solutions

[2] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction*, 22(2):7:1–7:35, Mar. 2015c. ISSN 1073-0516. URL `http://doi.acm.org/10.1145/2699751`

and represents each cluster as a synthesized solution that implicitly describes both the cluster center and its boundaries. Teachers can filter and further cluster solutions based on various criteria. One of OVERCODE's key abstractions is equating variables across solutions based on behavior during execution; these common variables are each renamed to their most popular student-given name, which highlights remaining differences in algorithms and syntax. In user studies, OVERCODE allowed teachers to more quickly develop a high-level view of student understanding and misconceptions, and provide feedback that is relevant to more student solutions. A few teachers can give personalized composition feedback on code from over a thousand students in a few hours. The feedback can be automatically reused in following semesters.

Other deployed systems: I developed and deployed DEAR BETA and DEAR GAMMA [3] in MIT's large introductory computer architecture course for collecting and distributing student-written debugging and optimization hints for the entire spectrum of student-constructed digital circuits. In the same class, I deployed MUDSLIDE, [4] the system I developed at Microsoft Research for collecting and visualizing the distribution and content of student confusion across the slides of a presentation *(CHI Honorable Mention)*. I also supervised an MIT EECS M.Eng. student who created and deployed GROVERCODE [5] for more quick and consistent exam grading for one of MIT's large introductory Python programming classes.

Generalizing to code in the wild

Understanding the space of possible code solutions is helpful beyond the classroom, as well.

Figure 4: In this screenshot, EXAMPLORE shows the head of the canonicalized distribution of `FileInputStream` API usage in 100 open-source Github repositories.

In collaboration with software engineering researchers at UCLA, we created EXAMPLORE (Fig. 4), an interactive visualization created by mining hundreds of thousands of open-source Github repositories

[3] E. L. Glassman, A. Lin, C. J. Cai, and R. C. Miller. Learnersourcing personalized hints. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, pages 1626–1636. ACM, 2016. URL `http://doi.acm.org/10.1145/2818048.2820011`

[4] E. L. Glassman, J. Kim, A. Monroy-Hernández, and M. R. Morris. Mudslide: A spatially anchored census of student confusion for online lecture videos. In *Proceedings of the Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1555–1564. ACM, 2015b. URL `http://doi.acm.org/10.1145/2702123.2702304`

[5] S. Terman. Grovercode: code canonicalization and clustering applied to grading. Master's thesis, Massachusetts Institute of Technology, 2016

to reveal the common and uncommon ways in which the open-source developer community uses a Java API method. [6]

To register hundreds of API usage examples against each other and display them overlaid, I designed a general abstraction, the API skeleton, grounded in (1) how API design is taught in software engineering curricula and (2) how API mining researchers conceptualize their task. The resulting interactive visualization reveals the distribution of canonicalized statements and structures enclosing the API method of interest. In a within-subjects lab study, we found that EXAMPLORE helped users understand the usage of unfamiliar APIs and answer common API usage questions accurately.

Now that we have shown that code demography generalizes beyond the classroom, there are a whole new set of users and tasks that can be supported. By continuing my collaboration with researchers in software engineering, I plan to explore the use of code demography to support (1) API designers who want to see how people use their API, (2) programmers who need to learn about how an unfamiliar function is used in practice, and (3) software engineers interested in a statistical picture of how the code base at their company is written before contributing their own code or participating in a code review.

[6] E. L. Glassman*, T. Zhang*, M. Hearst, B. Hartmann, and M. Kim. Visualizing api usage examples at scale. In *Proceedings of the Annual ACM Conference on Human Factors in Computing Systems*, CHI '18. ACM, 2018

Teaching Humans and Machines with Examples

Teaching humans

Variable names are an important component of code composition, and yet it is prohibitively time-consuming to comment on each student's variable names at scale. FOOBAZ [7] uses OVERCODE's common variable abstraction to reveal the distribution of student-chosen names for each common variable. Teachers curate and label interesting choices of good and bad names from the distribution, and FOOBAZ sends each student a set of these teacher-labeled examples to evaluate as alternative names for a variable in their own program. Students then compare their judgments to teacher labels to help train their inner variable naming critic.

[7] E. L. Glassman, L. Fischer, J. Scott, and R. C. Miller. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 609–617. ACM, 2015a. URL `http://doi.acm.org/10.1145/2807442.2807495`

Teaching machines

FIXPROPAGATOR [8] **allows teachers to teach the machine, by demonstration, how they would fix a bug** in a particular student solution. In the backend, a state-of-the-art program synthesis technique infers more general abstract syntax tree (AST) transformations, e.g., `var*var` becomes `f(var)*var`, that are consistent with the teacher's demonstration. [9] These inferred abstract transformations are applied to fix other buggy student solutions as well.

MISTAKEBROWSER, published with FIXPROPAGATOR, uses the same program synthesis technique to infer abstract code transformations from examples of students fixing bugs in their own submissions, which were mined from a massive programming course's autograder logs. In

[8] A. Head*, E. L. Glassman*, G. Soares*, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 89–98. ACM, 2017. URL `http://doi.acm.org/10.1145/3051457.3051467`

[9] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415. IEEE Press, 2017. URL `https://doi.org/10.1109/ICSE.2017.44`

other words, **as students fix their own bugs, the machine is learning to fix other student bugs in the same way**.

In both systems, incorrect submissions are clustered by the machine-inferred transformations that correct them and are presented back to the teacher as a high-level view of student bugs. For each cluster, teachers write feedback that can be propagated to all other current and future code submissions that can be fixed by the same transformation.

To explore the interpretability of another example-based inference algorithm, a collaborator's interactive latent variable model, I built an interface on top of the model to test how well domain experts could collaborate with the machine—by choosing examples and critical features—to define statistically valid clusters that were also relevant to the expert's tasks. [10]

[10] B. Kim, E. L. Glassman, B. Johnson, and J. Shah. ibcm: Interactive bayesian case model empowering humans via intuitive interaction. MIT CSAIL Technical Report, 2015

Research Agenda

The class of problems I work on—where people must answer questions that require a population-level characterization of relevant data—is vast. I have already demonstrated the value of code and data demography for programmers in the wild and in large programming classrooms. Next, I plan to create systems that help scientists, data wranglers, and non-programming end-users work with large data corpora.

How will people work with large corpora of complex data in the future?

Many people who work with data have datasets that are too large, too noisy, too complex, or too unstructured to make sense of at once. How can code demography be generalized to more kinds of data for which standard information visualization techniques do not work well? For example, a social scientist asked me to analyze a large collection of geolocated tweets from Egypt to discover population-level changes in political expression over time. I found that this is a difficult task because there are different but interrelated 'frequency bands' of activity present in this kind of data. Any one time resolution for examining the data left important trends 'out of focus.' Given my prior work on wavelet-like filters for multi-resolution analysis of biomedical signals, [11] I have designed some exploratory prototypes of multi-resolution analysis for time-evolving text corpora. I am now working with several students at UC Berkeley to create a prototype that can be tested by social scientists at the Berkeley Institute of Data Science.

[11] E. L. Glassman. A wavelet-like filter based on neuron action potentials for analysis of human scalp electroencephalographs. *IEEE Transactions on Biomedical Engineering*, 52(11):1851–1862, 2005

How can data demography increase algorithmic transparency?

When a domain expert applies a function to their large dataset, regardless of whether they wrote that function themselves or it was inferred from data, it is difficult to verify that the function performed

as the expert intended on every data point. [12] I plan to build systems that address this challenge by generalizing data demography to reveal *population-level changes* in datasets induced by a function. This is particularly important for functions that are otherwise difficult to interpret, e.g., neural networks. While exposing the induced changes may reveal machine errors that temporarily lower the expert's confidence in the function's correctness, the demographic view should also reveal information that helps the expert debug.

Prior to entering the field of human-computer interaction, I created distance functions for planning dynamic movements for robots. [13], [14] As a faculty member, I hope to reconnect with the robotics community and explore possible collaborations. For example, how can data demography help roboticists debug—and end-users interact with and trust—autonomous systems?

[12] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani. Interactive program synthesis. *CoRR*, abs/1703.03539, 2017. URL `http://arxiv.org/abs/1703.03539`

[13] E. L. Glassman and R. Tedrake. A quadratic regulator-based heuristic for rapidly exploring state space. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 5021–5028. IEEE, 2010

[14] E. L. Glassman, A. L. Desbiens, M. Tobenkin, M. Cutkosky, and R. Tedrake. Region of attraction estimation for a perching aircraft: A lyapunov method exploiting barrier certificates. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2235–2242. IEEE, 2012

How will humans communicate their intent to machines in the future?

The critical, possibly latent, structural features that distinguish examples from each other help both humans and machines learn concepts and infer abstractions that generalize well. Examples are, therefore, a natural medium for communication between people and machines. Code and data demography strongly supports this kind of communication: it exposes the population-level characteristics of large datasets in a way that keeps concrete examples front and center instead of hiding them behind nodes in large graphs or other abstractions.

Some of the next systems I hope to build will combine example-based inference techniques with data demography to help users thoroughly examine the data in their corpus (1) before selecting examples to teach the machine and (2) while reviewing the machine-inferred results. In the short term, I expect that data demography will help users clarify their own intentions and curate better examples to teach machines, much like OVERCODE and FOOBAZ helped teachers understand the state of their massive classroom and pick better examples to address student misconceptions.

Finally, as powerful as communicating with machines by example, annotation, and demonstration can be, existing systems can fail in opaque and frustrating ways. How do we build systems that *explain their failure* to synthesize a program or fit a model that is consistent with user-provided examples? The program synthesis toolkit my systems currently use, the Microsoft PROSE SDK, requires an expert-designed grammar to perform well on new tasks and corpora. Rather than co-create abstractions in a fixed grammar, what if the machine and the end-user could co-create, or at least modify, the grammar itself? As a first step, I hope to create visualizations and interaction mechanisms that demystify the internal state of program synthesis engines when they fail. I am discussing this idea with my colleagues at Microsoft Research who are core developers of the PROSE SDK.