Interactive systems for code and data demography

Elena L. Glassman

December 21, 2017

I am a researcher in **human-computer interaction (HCI)**, I design, build and evaluate systems for code demography, i.e., comprehending and interacting with population-level structure and trends in **large code corpora**. These systems augment human intelligence by giving users a "useful degree of comprehension in a situation that previously was too complex."[1]

In my doctoral and postdoctoral work at MIT and UC Berkeley, I have used *program analysis and synthesis techniques, interactive inference algorithms, visualization principles, and theories from cognitive science* to build systems that allow people to complete existing large-scale code-related tasks more quickly and answer new questions that were previously prohibitively time-consuming to investigate (Figs. 1 and 2). For example, OVERCODE (Fig. 3) is now deployed at UC Berkeley, where teachers give code composition feedback to more than 1500 students in a few hours.[2] EXAMPLORE (Fig. 4) allows programmers, API designers, and researchers to answer questions about how API methods are actually used in the wild.[3] Now, as a fellow at the Berkeley Institute of Data Science, I am exploring how to generalize these methods beyond code to help data scientists, social scientists, journalists, and other end-users more easily work with large amounts of data and communicate their intent to machines using concrete examples.

The conceptual key to my approach is defining *task-relevant abstractions* through data-driven (1) user-centered design or (2) inference algorithms. For example, I designed EXAMPLORE's abstract API skeleton to register and align hundreds of usage examples against each other so that users can get a high-level view of a corpus without sacrificing the ability to read concrete code. This design is supported by theories of human learning, such as analogical learning and Variation Theory: showing multiple aligned examples simultaneously helps induce accurate abstractions in the user's mind. In FIXPROPAGATOR, the abstractions are inferred from data: as a programming teacher begins to fix and give feedback on buggy student code submissions, the back end infers more general, abstract code transformations to propagate fixes and relevant teacher feedback to other buggy student code.[4]

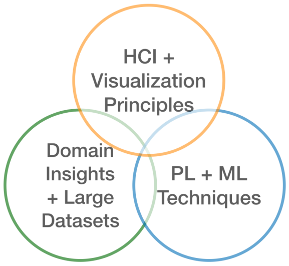

HCI + Visualization Principles
Domain Insights + Large Datasets
PL + ML Techniques

Figure 1: Common system components.

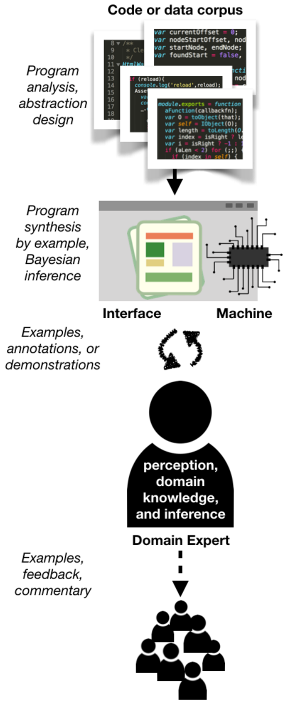

Code or data corpus
Program analysis, abstraction design
Program synthesis by example, Bayesian inference
Interface
Machine
Examples, annotations, or demonstrations
perception, domain knowledge, and inference
Domain Expert
Examples, feedback, commentary

Figure 2: Common system architecture.

1. *Tools for teachers and students in massive classrooms*

Enrollment in introductory programming and data science courses is skyrocketing. Even hardware design classes can contain hundreds of students, each constructing their own simulated circuits. 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 explore this space of possibilities.

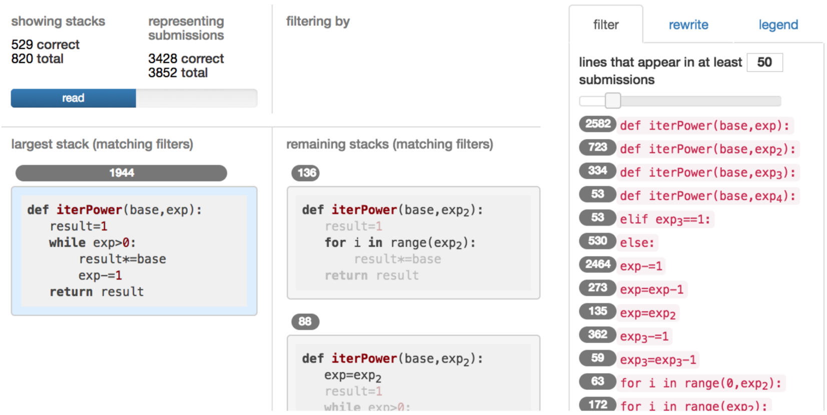
OVERCODE is an example of code demography for visualizing and exploring thousands of solutions to the same programming problem. It uses both static and dynamic analysis to cluster similar solutions and represents each cluster as a synthesized solution that implicitly describes both the cluster center and its boundaries. Teachers can filter and further cluster solutions based

[1] D. C. Engelbart. Augmenting human intellect: A conceptual framework. *Stanford Research Institute. Retrieved March*, 1: 2007, 1962

[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

[3] 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

[4] 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



showing stacks
529 correct
820 total

representing submissions
3428 correct
3852 total

filtering by

largest stack (matching filters)

1944

```
def iterPower(base,exp):
    result=1
    while exp>0:
        result*=base
        exp-=1
    return result
```

remaining stacks (matching filters)

136

```
def iterPower(base,exp2):
    result=1
    for i in range(exp2):
        result*=base
    return result
```

88

```
def iterPower(base,exp2):
    exp=exp2
    result=1
    while exp>0:
```

lines that appear in at least 50 submissions

```
2582 def iterPower(base,exp):
723  def iterPower(base,exp2):
334  def iterPower(base,exp3):
53   def iterPower(base,exp4):
53   elif exp3==1:
530  else:
2464 exp-=1
273  exp=exp-1
135  exp=exp2
362  exp3-=1
59   exp3=exp3-1
63   for i in range(0,exp2):
172  for i in range(exp2):
```

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.

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 on selected examples that are relevant to more student solutions.

Societal and potentially commercial impact

OVERCODE has now been fully integrated into the code composition feedback process of the largest introductory Python programming course at UC Berkeley, which regularly enrolls 1500 students: rather than distributing work to as many as 50 graders, a handful of teaching assistants can give feedback to the entire class in a few hours. We expect that this same feedback will be re-sent to students in future semesters, with little additional effort on the part of the teaching staff. Two Master's students, one at MIT and one at UC Berkeley, have earned—or soon will—their degree in EECS by contributing to this effort and a third student at UC Berkeley is leading the charge to make this tool available to all other schools with large Python programming courses. We are also fielding requests to help others adapt this technology to other programming languages.

Scalable, data-driven teaching by example

Variable names are an important component of code composition, but it is difficult to quickly get an accurate picture of how well all your students are naming variables, and prohibitively time-consuming to comment on individual variable names at scale. FOOBAZ solves this problem by letting teachers draw from their own students' naming choices to create a set of examples that clarify the concept of a good, contextually appropriate variable name.[5] Specifically, FOOBAZ uses OVERCODE's common variable abstraction to reveal the distribution of student-chosen names for each common variable. Teachers curate and label names of varying quality for the most common variable roles, and FOOBAZ sends each student a set of these examples to evaluate as alternative names for the corresponding variable role in their own program. Students then compare their judgments to teacher labels to help train their inner variable naming critic.

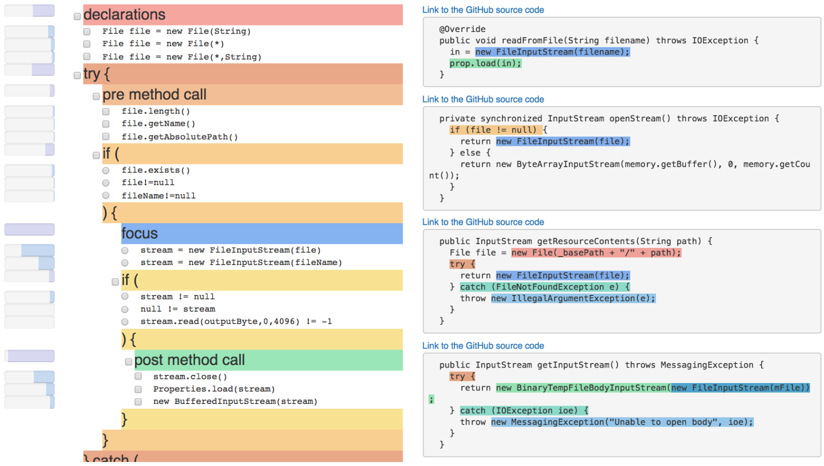
[5] E. L. Glassman, L. Fischer, J. Scott, and R. C. Miller. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 609–617. ACM, 2015a. URL http://doi.acm.org/10.1145/2807442.2807495

Additional systems

I also developed and deployed DEAR BETA and DEAR GAMMA 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.[6] In the same class, I deployed MUDSLIDE, 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)*.[7] I also supervised an MIT EECS M.Eng. student who created and deployed GROVERCODE, an extension of OVERCODE for more quick and consistent exam grading for one of MIT's large introductory Python programming classes.[8]

2. *Supporting programmers in the wild and at companies with large codebases*

Understanding the space of possible code solutions is helpful beyond the classroom, as well. EXAMPLORE (Fig. 4) is an example of code demography for visualizing and exploring thousands of code examples using the same API: in collaboration with software engineering researchers at UCLA, we created an interactive visualization by mining hundreds of thousands of open-source Github repositories to reveal the common and uncommon ways in which the open-source developer community uses a Java API method.[9] In a within-subjects lab study, we found that EXAMPLORE helped programmers understand the distribution of usage patterns of a particular API method in the wild, and answer common API usage questions accurately. I plan to make this tool available as an online resource that complements official documentation and Q&A sites. I also hope to work with engineers and researchers at companies with large proprietary codebases and APIs, like Google and Facebook, to customize this tool for their internal on-boarding and code review needs. Through initial contacts, it is clear that these companies are eager to adopt and exploit new methods and interfaces, like EXAMPLORE, to increase programmer productivity.



```
□ declarations
  □ File file = new File(String)
  □ File file = new File(*)
  □ File file = new File(*,String)
□ try {
  □ pre method call
    □ file.length()
    □ file.getName()
    □ file.getAbsolutePath()
  □ if (
    ○ file.exists()
    ○ file!=null
    ○ fileName!=null
  ) {
      focus
      ○ stream = new FileInputStream(file)
      ○ stream = new FileInputStream(fileName)
    □ if (
      ○ stream != null
      ○ null != stream
      ○ stream.read(outputByte,0,4096) != -1
      ) {
       □ post method call
         □ stream.close()
         □ Properties.load(stream)
         □ new BufferedInputStream(stream)
      }
  }
} catch (
```

Link to the GitHub source code

```
@Override
public void readFromFile(String filename) throws IOException {
  in = new FileInputStream(filename);
  prop.load(in);
}
```

Link to the GitHub source code

```
private synchronized InputStream openStream() throws IOException {
  if (file != null) {
    return new FileInputStream(file);
  } else {
    return new ByteArrayInputStream(memory.getBuffer(), 0, memory.getCou
nt());
  }
}
```

Link to the GitHub source code

```
public InputStream getResourceContents(String path) {
  File file = new File(_basePath + "/" + path);
  try {
    return new FileInputStream(file);
  } catch (FileNotFoundException e) {
    throw new IllegalArgumentException(e);
  }
}
```

Link to the GitHub source code

```
public InputStream getInputStream() throws MessagingException {
  try {
    return new BinaryTempFileBodyInputStream(new FileInputStream(mFile))
;
  } catch (IOException ioe) {
    throw new MessagingException("Unable to open body", ioe);
  }
}
```

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

[6] 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`

[7] 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`

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

[9] 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

3. Collaborating with PL and ML researchers to improve the interface between people and intelligent back ends

Theories of human concept learning, e.g., Variation Theory,[10] assert that showing people strategically diverse sets of examples will help them construct mental abstractions that generalize well. The same is true when teaching machines. In the following projects, I show how examples are mined for data-driven teaching of both humans and machines. To achieve these results, I collaborated with academic and industry researchers in programming languages (PL) and machine learning (ML) to produce systems with cutting-edge intelligent back ends and front ends that enable real-world impact.

In my first exploration of example-based inference frameworks, specifically the interactive Bayesian Case Model (iBCM), I collaborated with iBCM's creator to build 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.[11] These experts were Python teachers clustering student-written Python programs and, despite a clever encoding of these programs that captured some semantics, iBCM was not well matched to the task: it was only concerned with statistical distributions of features, and therefore not sufficiently concerned with the semantics of the programs themselves.

When I joined UC Berkeley as a postdoctoral scholar funded by the NSF Expeditions in Computer Augmented Program Engineering (ExCAPE) grant, I was introduced to program synthesis by example and the newly available Microsoft PROgram Synthesis by Example (PROSE) SDK. I collaborated with PL researchers to build two successful systems on top of this technology (FIXPROPAGATOR and MISTAKEBROWSER, described below) and co-organized a workshop to help other researchers use PROSE for their own research purposes. At the Dagstuhl Seminar "Approaches and Applications of Inductive Programming," we concluded that the tools are mature enough for us to concentrate on developing theory and methods around *how users interact* with them.

Interacting with program synthesis by example

I led the development of two interactive systems that leverage program synthesis by example.[12] FIXPROPAGATOR **allows teachers to teach the machine, by demonstration, how they would fix a bug** in a particular student solution. In the back end, 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.[13] These inferred abstract transformations are applied to fix other buggy student solutions as well. MISTAKEBROWSER 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 the autograder logs of a massive programming course. In other words, **as students fix their own bugs, the machine is learning to fix other student bugs in the same way**. Incorrect submissions are clustered by the machine-inferred transformations that correct them and presented to the teacher. As a result, the space of common and uncommon bugs becomes explicit and human-understandable almost at a glance. For each cluster, teachers write feedback that can be propagated to all current and future code submissions fixed by the same transformation.

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

[11] 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

[12] 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`

[13] 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`

Research directions

Generalizing from code to data demography

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, especially data types that have structure but also require reading and interpretation, e.g., natural language or mathematical statements written in LaTeX? As a first step, I would like to investigate whether the types of abstractions inferred by data-driven program synthesis algorithms can help extend visualizations like EXAMPLORE to represent large semi-structured collections of text, e.g., massive log files or large collections of paper abstracts.

There are also ample opportunities to assist data journalists and social scientists. For example, a professor of international relations at Harvard University asked me to analyze a large collection of geolocated tweets from Egypt to discover population-level changes in political expression over time during the recent revolution. I discovered that the same trade-offs between time and frequency resolution in signal processing existed in the processing of time-varying textual corpora. Given my prior work on wavelet-like filters for multi-resolution analysis of biomedical signals,[14] I am now composing algorithms and visualizations for multi-resolution analysis for time-evolving text corpora. I hope to create and test prototypes in collaboration with a variety of domain experts who could benefit from these tools.

How can we more naturally communicate our intent to machines?

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 support 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 systems I hope to build next 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. I expect that data demography will help users clarify their own intentions[15] 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. I hope to continue working with experts in programming languages and machine learning to improve the ways in which humans and machines communicate and collaborate, in order to develop applications with real-world impact.

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 learned from data, it is difficult to verify that the function performed as the expert intended on every

[14] 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

[15] 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`

data point. Machine-inferred functions can also be difficult to inspect directly: they may be composed of many layers, nonlinearities, and weights, a hyperplane in high-dimensional space, or a program written in a grammar that the end-user is completely unfamiliar with. 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, e.g., quickly identify false positives and false negatives or data in the corpus that was incorrectly extracted or transformed.

For a concrete, near-term example of how data demography could help users review machine-inferred results, consider how MISTAKEBROWSER currently communicates a machine-induced bug-fixing abstract code transformation to the user, a teacher: the interface renders a list of all the buggy student code submissions before and after the bug-fixing transformation was applied. It can be cognitively exhausting to review this list, so the teacher often only looks at a few transformed student code submissions and does not necessarily get an accurate picture of how the transformation is acting on all their students' code. Teachers may form inferences more quickly and accurately if each cluster of transformed code is visualized in an aligned and overlaid way, much like the API usage examples in EXAMPLORE.

Similarly, I would like to explore how data demography can increase the transparency of—and trust in—autonomous system behavior. Just prior to entering the field of HCI, I was publishing distance functions for planning dynamic movements for robots.[16],[17] Just as code demography helps programmers quickly explore an unfamiliar API, can *trajectory demography* help roboticists or end-users quickly make correct inferences about how a new robot will behave under a variety of circumstances? As a faculty member, I hope to reconnect with the robotics community and explore possible collaborations.

When the system fails, how do we debug?

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 that is consistent with user-provided examples? A simple failure mode in program synthesis occurs when the synthesizer's grammar does not match the user's mental model: perhaps the user is teaching the machine how to extract something from raw HTML but the grammar has no notion of matching start and end tags. 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 will continue collaborating with programming language researchers in academia and industry, such as my colleagues on the PROSE SDK core developer team at Microsoft Research, to create better inspection, debugging, and interaction tools for end-users and developers alike.

[16] 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

[17] 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