Chapter 1

Related Work

Systems that help students in massive programming courses may build on work from program analysis, program synthesis, crowd workflows, user interface design, machine learning, intelligent tutoring systems, natural language processing, data mining, and learning science. This chapter summarizes relevant technical and psychological work that supports the pedagogical value of the systems developed in this thesis and gives context to the technical contributions.

1.1 Clustering

Grouping similar items, i.e., *clustering*, is a fundamental human activity for organizing, making sense of, and drawing conclusions from data. Across many scientific fields, clustering goes by different names but serves a common purpose: helping humans explore, interpret, and summarize data [**?**].

~~This~~ Clustering is *unsupervised machine learning*. There are no labels, no ground truth. For example, is there one true clustering of all the articles the New York Times ever published?

Humans have goals, and how useful clusters are to them is all that matters.

~~Humans are pattern finders.~~ A human can look at collections of items and group them in various ways. For example, a collection of dogs could be grouped according to their size or the color of their fur. Both clusterings are equally valid~~. Clustering by size may be more useful if the purpose is to decide which dogs get walked together~~, but if a dog walker is deciding which dogs to walk together, clustering by size is more useful.

~~Computers need more instructions before they can step in and help cluster items, when there are too many for a human to cluster them alone. Items need to be represented in a machine-readable way, and human judgements need to be replaced with computation.~~

~~The first critical decision when preparing a collection for clustering by a computer is how to represent the itemsin a machine-readable format, e. g. , as images or sets~~ In order for computers to cluster items, the system designer needs to choose a representation for the items, possibly select or extract features from that representation, choose a clustering method, and assess the validity of its results. The chosen *representation* effects every subsequent step of the clustering process [**?**]. The representation might be an image or a set of attributes, also called *features*, like weight, height, and fur color. ~~This representationeffects every subsequent step of the clustering process [**?**].~~ Since the clustering is performed for a human with a goal, the representation ideally captures aspects of the items that are relevant to that goal.

Further customizing the item representation for the human goal, also known as *feature extraction*, is optional but often very helpful. Feature extraction is computing new features about each item from the original representation. These new features may better capture the aspects of items that are relevant to the human goal. For example, if the goal is to partition dogs by fur color and dogs are represented by images, one might extract the most common color in every dog image.

A second optional step is *feature selection*, the process of determining what features can be ignored because they are irrelevant to the human goal. For example, if the goal is to partition dogs by fur color and dogs are represented by collections of features, then only the fur color feature may be selected, ignoring height and weight.

Hopefully, ~~at the end of this process~~after choosing a representation and possibly performing feature selection and extraction, some items are closer to each other than they are to others, with respect to some aspect of the items the human cares about. If a computer is to verify this, one needs to define exactly what the *closer* means. The computer needs a human to define a *distance function* that quantifies how close or far items are from each other.

The next step is choosing a *clustering method*~~, i. e., a general strategy.~~ . Clustering is finding clusters of items that are closer to each other than they are to items in other groups. There is no one best method for all clustering problems, but some methods produce clusters that support the human's problem-specific goal better than others. Many methods require the human to provide additional parameters, such as how similar two items need to be to be grouped together or how many clusters to look for.

Some methods have an *objective function* that quantifies some desirable characteristic of clusters to maximize or some undesirable characteristic of clusters to minimize [**?**]. One objective function sums up how different each item is from its cluster center. Presumably, the smaller the differences, the better the clustering. Specific clustering algorithms may define different centers for each cluster, tally up the differences in different ways, and follow different processes to minimize the objective function.

Some methods are more easily characterized by their process than their objective function. For example, hierarchical clustering methods are characterized as either:

- *agglomerative* or bottom-up, by merging individual items into small clusters, then merging small clusters into larger clusters, etc., or
- *divisive* or top-down, by splitting the entire collection of items into clusters, then splitting those clusters into further clusters, until all clusters only contain one item [**?**].

The objective functions used in these methods help determine which clusters to merge or split.

Some statistical clustering methods, such as mixture models, assume that the items are

samples from underlying, unseen distributions of items. By making some assumptions about what those distributions might look like, an algorithm can map items to clusters to maximize a statistical measure of fit between the observed data and the unobserved distributions.

~~Many methods require the human to provide additional values and thresholds, such as how similar two items need to be to be grouped together or how many clusters to look for. Some methods incorporate techniques that anticipate outliers and attempt to be robust to them.~~

The types of clusters produced by clustering algorithms can be partitioned in several ways [**?**], such as:

- *Hard* versus *soft*: If each item is mapped to a single cluster, the method is hard. If items can partially belong in different degrees to multiple clusters, the method is soft.
- *Partition* vs. *overlap* vs. *hierarchy*: If each item belongs to one and only one cluster, the method is producing strict partitions. If each item belongs to one or more clusters, than the method is producing overlapping clusters. If each item belongs to one cluster and clusters are strictly contained by other clusters, the method is hierarchical.

Some methods do not produce clusters of items, but rather clusters of components within items. For example, Latent Dirichlet Allocation (LDA) [**?**] does not cluster entire documents. It produces a hard, strict partition of words into different clusters, typically called *topics*. At the level of documents, however, this can look like a soft clustering: an individual document might be 50% hospital-related words, 30% government-related words, and 20% negotiation-related words.

Clusters are hard to objectively evaluate. It is expensive and time consuming to bring in humans with goals to evaluate how useful the clusters are to them. *Cluster validation* refers to the metrics used to approximate the quality of clusters with little or no human input [**?**]. These metrics can be the basis of choosing one method over another, one parameter value over another, or one clustering over another quickly but approximately. Note that a clustering may have good quality scores with respect to those cluster validation metrics, but if it is difficult for the human to understand what each cluster contains, they may have

trouble using it to achieve their goals. Therefore, there are a growing number of methods that are designed with *interpretability* in mind.

Another way to produce more helpful clusterings is to involve the human in an *interactive* process. The clustering algorithm produces clusters and gives the human one or more ways to give feedback. Feedback could be in the form of indicating what is good or bad about this clustering. Feedback could be the human reaching in and directly moving an item from one cluster to another. The method reruns, based on this new information, and produces a new clustering for the human to evaluate and give feedback on.

~~One aspect of clustering that does not fall into the traditional conversation about features, distance functions, and methods is interpretability. A clustering may have good quality scores with respect to those cluster validation metrics, but if it is difficult for the human to understand what each cluster contains, they may have trouble using it to achieve their goals.~~

There are thousands of published ~~methods that exhibit~~ papers and working systems that use combinations of these properties and strategies. When comparing ~~methods~~two systems that cluster, it is helpful to break the comparison down by representation, features, distance function, objective function, and algorithm they use as well as the type of clusters they produce. In subsequent sections, this language will be used to help describe the many different code analysis and clustering processes found in related work.

1.2 Mining Solution Variation

One important aspect of both engineering and design is that there are multiple ways to solve the same problem, each with their own trade-offs. The consequences of these design choices can be significant. Suboptimal solutions, including poorly designed code, can have high personal, safety, and economic costs.

It may be possible to learn good and bad practices from large collections of previous

solutions by looking at common and uncommon choices and making judgements about their successes and trade-offs. In order to understand the space of current solutions, there are many questions one could ask. In the face of a common design choice, what do designers most commonly pick? Has this changed over time? What are popular design alternatives? What are examples of design failures that should be learned from and never repeated? What are examples of design innovations that are clearly head-and-shoulders above the rest? The answers to these questions could fuel education about designing solutions that complies with variation theory recommendations. OverCode, Foobaz, and the comparison learnersourcing workflow try to answer these questions for student solutions written in code.

1.2.1 Code Outside the Classroom

Web pages

Ritchie et al. [**?**] describe a user interface for finding ~~helpful, i.e., relevant or inspiring,~~ design examples from a curated database of web pages. Their work is intended to support designers who like to refer to or adapt previous designs for their own purposes. Traditional search engines only index the content of web pages. Their system indexes web ~~pages'~~ page design style by automatically extracting global stylistic and structural features from each page. Instead of manual browsing, users can search and filter a gallery of design-indexed pages. Users can provide an example design in order to find similar and dissimilar designs, as well as high-level style terms like "minimal."

Kumar et al. [**?**] defined *design mining* as "using knowledge discovery techniques to understand design demographics, automate design curation, and support data-driven design tools." Their work goes beyond searching and filtering a gallery of hundreds of curated web pages. Their Webzeitgeist design mining platform allows users to query a repository of hundreds of thousands of web pages based on the properties of their Document Object Model (DOM) tree and the look of the rendered page. A vector of descriptive features is computed for each DOM node in each page.

Webzeitgeist enables users to ask and answer questions like the following, with respect to a large web page repository. What are all the distinct cursors? What are the most popular colors for text? How many DOM tree nodes does a typical page have? How deep is a typical DOM tree? What is the distribution of aspect ratios for images? What are the spatial distributions for common HTML tags? How do web page designers use the HTML canvas element?

To dig into examples of a particular design choice, users can, for example, query for all pages with very wide images. The result is a set of horizontally scrolling pages. Alternatively, users can query for web pages that have a particular layout, like a large header, a navigational element at the top of the page, and a text node containing greater than some threshold of words, in order to see all the examples of pages that fit those layout specifications. Specific combinations of page features can imply high-level designs as well so, with careful query construction, users can query for high-level ideas. For example, querying for pages with a centered HTML input element and low visual complexity retrieves many examples that look like the front pages of search engines.

Android Apps

Shirazi et al. [**?**] and Alharbi and Yeh [**?**] describe automated processes for taking apart and analyzing Android app code as well as empirical analyses of corpora of Android apps available from the Google Play app store. Shirazi et al. analyzed the 400 most popular free Android applications, while Alharbi and Yeh tracked over 24,000 Android apps over a period of 18 months. Alharbi and Yeh captured each update within their collection window, as well. They decompiled apps into code from which UI design and behavior could be inferred, e.g., XML and click handlers, and tracked changes across versions of the same app. Both papers analysed population-level characteristics of their corpora, answering questions like: what is the distribution of layout design patterns, among the seven standard Android layout containers? What are the most common design patterns for navigation, e.g., tab layout and horizontal paging? Have any apps switched from one pattern to another? How quickly are

newly introduced design patterns adopted? What are the most frequent interface elements? And combinations of interface elements? How many applications does that combination cover?

Open-Source Code Repositories

Ideally, code is not just correct, it is simple, readable, and ready for the inevitable need for future changes [**?** **?**]. How can we help students reach this level of programming composition zen? How can we learn from others' code, even after we become competent, or even an expert, at the art of programming?

For the same reason we look at patterns in design across web pages and mobile apps, we can look at the design choices already made by humans who share their programs. Rather than using web crawlers or app stores, we can process millions of public repositories hosted online. What can we learn about good and bad code design decisions from these collections?

These code analysis techniques that follow are most closely related to those developed for OverCode and Foobaz. The OverCode and Foobaz pipelines are optimized for user interfaces that help users answer design mining questions about their student solutions and put the code front and center.

Code, like natural language, exhibits regularity. Hindle et al. [**?**] argue that the regularity of code may be even more true for code than for natural language. Allamanis and Sutton [**?**] observe that some syntactic fragments serve a single semantic purpose and recur frequently across software projects. Fast et al. [**?**] observe that poorly written code is often syntactically different from well written code, with the caveat that not all syntactically divergent code is bad.

Mining Idioms

Idiomatic code is written in a manner that experienced programmers perceive as normal or natural. Idioms are roughly equivalent to mental chunks, i.e., the memory units characterized

by George Miller [**?**]. I will borrow an example from Allamanis and Sutton: [**?**]

- `for (int i=0;i<n;i++) ...` is a common idiom for looping in Java.
- do-while and recursive looping strategies are not.

Fast et al. [**?**] break the definition of idioms into two levels. An example of a high-level idiom is code that initializes a nested hash. An example of a low-level idiom is code that returns the result of an addition operation. Some languages support a variety of different, equally good ways to do the same thing. Others encourage a single, idiomatic way to achieve each task.

Idioms can and do recur throughout distinct projects and domains (unlike repeated code nearly verbatim, i.e., clones) and commonly involve syntactic sugar (unlike API patterns). In general, clone detectors look for the largest repeated code fragments and API mining algorithms look for frequently used sequences of API calls. Idiom mining is distinct because idioms have syntactic structure and are often wrapped around or interleaved with context-dependent blocks of code.

There are enough idioms for some languages that they have lengthy, highly bookmarked and shared online guides. StackOverflow has many questions asked and answered about the appropriate language or library-specific idioms for particular, common tasks. It is difficult for expert users of each language or library to catalogue all the idioms. It is much more practical to simply look at how programmers are using the language or library and extract idioms from the data.

Hindle et al. [**?**] used statistical language models from natural language processing to identify idiom-like patterns in Java code. They found that n-gram language models built by analyzing ~~corporacaptured~~ corpora captured a high level of project- and domain-specific local regularity in programs. Local regularities are valuable for statistical machine translation of natural language. They may prove useful in analogous tasks for software as well. For example, the authors trained and tested an n-gram model token suggestion engine that looks at the previous two tokens already entered into the text buffer and predicts the next one the

programmer might type.

Allamanis and Sutton [**?**] automatically mine idioms from a corpus of idiomatic code using nonparametric Bayesian tree substitution grammars. The mined idioms correspond to important programming concepts, e.g., object creation, exception handling, and resource management, and are often library-specific. They found that 67% of the idioms mined from one set of open source projects were also found in code snippets posted on StackOverflow.

Fast et al. [**?**] computed statistics about the abstract syntax trees (ASTs) of three million lines of popular open source code in the 100 most popular Ruby projects hosted on Github. AST nodes are normalized, and all identical normalized nodes are collapsed into a single database entry. The unparsed code snippets that correspond to each normalized node are saved. Codex normalizes these snippets by renaming variable identifiers, strings, symbols, and numbers to `var0`, `var1`, `var2`, `str0`, `str1`, etc. Note that this fails when primitives, like specific strings and numbers, are vital to interpreting the purpose of the statement.

The resulting system, Codex, can warn programmers when they chain or compose functions, place a method call in a block, or pass an argument to a function that is infrequently seen in the corpus. It is fast enough to run in the background of an IDE, highlighting problem statements and annotating them with messages like, “We have seen the function split 30,000 times and strip 20,000 times, but we’ve never seen them chained together.” Codex can be queried for nodes by code complexity; type, i.e., function call; frequency of occurrence across files and projects; and containment of particular strings.

Mining Larger Patterns in Code

In the code of working applications, Ammons et al. [**?**] observed that common behavior is often correct. Based on that observation, they use probabilistic learning from program execution traces to infer the program’s formal correctness specifications. Inferring formal specifications for programs is valuable because programmers have historically been reluctant to write them. During program execution, the authors summarize frequent patterns as state machines that can be inspected by the programmer. As a result, the authors identified correct

protocols and some previously unknown bugs.

Buse and Weimer [**?**] go beyond idioms to mining API usage. Based on a corpus of Java code, they find examples that reference a target class, symbolically execute it to compute intraprocedural path predicates while recording all subexpression values, identify expressions that correspond to one use of the class, capture the order of method calls in those concrete examples, then use K-medoids to cluster these extracted concrete use examples with a custom formal parameterized distance metric that penalizes for differences in method ordering and type information. Concrete use examples within the same cluster are merged into abstract uses represented as graphs with edge weights that correspond to counts of how many times node X happens before node Y. Finally, they have a synthesis method to express these abstract use graphs in a human-readable form, i.e., representative, well-formed, and well-typed Java code fragments.

Mining Names

Without modifying execution, names can express to the human reader the type and purpose of an object, as well as suggest the kinds of operators used to manipulate it [**?**]. Perhaps as a direct result, variable names can exhibit some of the same regularity exhibited by code, in general. Høst and Østvold [**?**] go so far as to call method names a restricted natural language they dubbed Programmer English.

Høst and Østvold [**?**] ran an analysis pipeline on a corpus of Java that performs semantic analysis on methods and grammatical analysis on method names. It generates a data-driven phrasebook that Java programmers can use when naming methods. In a second publication [**?**], they formally defined and then automatically identified method naming bugs in code, i.e., giving a method a name that incorrectly implies what the method takes as an argument or does with an argument.

They did this by identifying prevalent naming patterns, e.g., contains-*, which occur in over half of the applications in the corpus and match at least 100 method instances. They also determined and cataloged the attributes of each method body, such as whether it read or

wrote fields, created new objects, or threw exceptions. If almost all the methods whose names match a particular pattern, e.g., contains-*, have an attribute or do not have some other attribute, it is automatically determined to be an implementation rule that all names in the corpus should follow. When run on a large corpus of Java projects, this analysis pipeline found a variety of naming bugs.

Fast et al.'s Codex [**?**] produced similar results. By keeping track of variable names in variable assignment statements, it can warn programmers when their variable name violates statistically established naming conventions, such as the (probably confusing) naming of a hash object "array."

Foobaz can go beyond Fast et al. [**?**] and Høst and Østvold's [**? ?**] work in providing feedback on variable names because all solutions are known to address the same programming problem.

1.2.2 Code Inside the Classroom

The common purpose of the code in a corpus of student solutions to the same programming problem almost certainly contributes to the regularity already found in code from large corpora of open source projects. However, student-written code may not exhibit the same regularity as code written by software developers contributing to an open source project. In other words, the regularity of student solutions to the same programming problem that comes from sharing a common purpose may be counter-balanced by the variety of student coding styles.

It is easier to run student solutions than code in open source projects. Teacher-designed tests already exist, some available to and some hidden from students while they developed their solutions, and some generated on the fly through fuzz testing [**?**]. This means that dynamic analysis can complement the static analysis of solutions.

Also unlike corpora of open source code, the solutions in large corpora of student code

often require feedback, so that the author can learn. Automated feedback on solutions to programming problems is still an area of active research. For example, assigning grades based solely on the number of teacher-designed tests the code passes may not capture what teachers care to grade on. A single error in a solution can cause a near-perfect solution to fail all test cases. A solution can perform perfectly on test cases but be poorly written or violate instructions, e.g., use the wrong algorithm to achieve the right results. The test cases themselves may be poorly designed. Some teachers hand-review hundreds of exam solutions because the autograder output is not sufficient to assign grades.

When a course has hundreds or thousands of students, it can be challenging to provide feedback to each solution quickly and consistently by hand. Design mining techniques and interfaces can help teachers explore and understand the whole space of solutions as well as distribute feedback to specific subsets of solutions, or subsets within solutions. It could be an important tool to assist in the sometimes difficult, manual task of identifying pedagogically valuable examples for illuminating a concept or principle. Distributing feedback can also be approached as an unsupervised or supervised, possibly interactive, machine learning problem, by leveraging clustering, classification, clone detection, and mixture modeling methods.

Regularizing Student Solutions

Solutions to the same problem can have different syntax but common semantics. Semantics-preserving transformations can be used to identify and regularize semantically equivalent code. This can include standard compiler optimizations, such as dead code removal, constant folding, copy propagation, and the inlining of helper functions [**?**]. It can also include transformations, like changes in the order of operands to a commutative function, that make one solution closer to another solution with respect to some definition of edit distance. Applying semantics-preserving transformations, sometimes referred to as normalization or standardization, has been used for a variety of applications, including detecting clones [**? ?**], diagnosis of bugs in student-written programs [**?**], and self-improving intelligent tutoring

systems [**?**].

Semantics-preserving transformations will not change the *schema(s)* within a solution. A schema is a high-level cognitive construct by which humans understand or generate code to solve problems [**?**]. For example, two programs that implement bubble sort have the same schema, bubble sort, even though they may have different low-level implementations.

Nguyen et al. [**?**] mines probabilistic semantically equivalent AST subtrees from a corpus of solutions to the same problem. The equivalences are probabilistic because the subtrees are only verified to be semantically equivalent with respect to the problem and the specific test cases provided. ~~In future work, this could also be used for regularization of code. For example, all subtrees in all solutions could be replaced with the most popular subtree that is probabilistically semantically equivalent to it, much like OverCode renames variables to their most popular names after determining that they are probabilistically semantically equivalent~~OverCode uses a similar technique for identifying common variables across solutions.

In OverCode and Foobaz, variables are carefully tracked and strategically renamed, rather than replaced by generic placeholders. This normalization step is novel in that its design decisions were made to maximize *human readability* of the resulting code. As a side-effect, syntactic differences between answers are also reduced.

Features and Distance Functions for Student Solutions

Solutions can be represented as sets or vectors of hand-crafted computed features, sequences of tokens, or graphs. In the context of software, tokens could be characters or tokens corresponding to the lexical rules of the programming language. Drummond et al. [**?**] catalog additional distance measures that are potentially helpful for clustering interactive programs.

Just as the authors of Webzeitgeist defined sets of features to be computed for each node in a web page DOM [**?**], there are many sets of features used to estimate program similarity

in the literature. Aggrawal et al. [**?**], Elenbogen and Seliya [**?**], Roger [**?**], Rees [**?**], Huang et al. [**?**], Kaleeswaran et al. [**?**], and Taherkhani et al. [**?**] each defined a set of features that could be computed automatically for each solution and represented as a numerical feature vector. These feature sets each include static or dynamic features. Some include both.

Static features include counts of various keywords and tokens in the solution, e.g., control flow keywords, operators, constants, and external function calls; counts of each type of expression in the solution; measures of code complexity; length of commented code; scores from linting scripts; function name lengths; line lengths; and entire solution lengths. To quantify the goodness of a solution's style, AutoStyle [**?**] used the ABC score, a weighted count of assignments, branches, and conditional statements in a block of code.

Dynamic features include data collected from running a solution on each test case, e.g., the solution output and whether or not it is correct with respect to the teacher specification, the evolution of values assigned to each variable within the solution, and the order in which statements in the solution are executed. For example, Huang et al. [**?**] create an output vector for each solution: a binary vector representing the solution's correctness on each test case. For approximately one million solutions to 42 programming problems collected from the original Machine Learning MOOC offered by Stanford, the authors found that a teacher could cover 90% of solutions or more in almost all problems by annotating the top 50 output vectors. However, the authors acknowledge that many different mistakes can produce the same output vector.

Variable behavior is a specific kind of dynamic feature that merits additional description. Just as there is evidence that experts subconsciously internalize *control flow plans*, like the Running Total Loop Plan that accumulates partial totals, there is evidence that experts subconsciously internalize *variable plans* [**?**]. Variable plans are characterized by the function or role that it serves in a function, how it is initialized and updated, and conditional statements on the variable's value.

Empirically, the number of distinct variable roles found in introductory-level programs

is small. While reviewing three introductory programming textbooks written for Pascal, Sajaniemi [**?**] hand-labeled the role of variables within each provided example program, based only on the pattern of successive values each variable took on. Nine variable roles, e.g., "stepper" and "one-way flag", covered 99% of the variables in all 109 programs found in those textbooks. Sajaniemi notes that a single variable can switch roles during execution, properly or sporadically. A proper switch is when its final value while serving in one role is its initial value when serving in the next role. A sporadic switch is one in which the variable is reset to a new value at some point, to serve a new role that may or may not have anything to do with its previous role. Sajaniemi has been credited for creating what is now known in the literature as role of variables theory.

Further work independently confirms and operationalizes Sajaniemi's insights. Taherkhani et al. [**?**] found that 11 variable roles cover all variables in novice-level programs, including object-oriented, procedural, and functional programming styles, and went further to automatically classify algorithms based on variables and some additional features. Gulwani et al. [**?**] also depend on variable behavior to recognize different algorithmic approaches. They ask teachers to annotate source code, by hand, with key values that differentiate algorithmic approaches.

OverCode and Foobaz make use of a pipeline that characterizes each solution as (1) a set of variables, which are distinguished from each other by their dynamic behavior during execution on test cases, (2) a set of one-line statements normalized in a novel way by those identified variables and (3) the solution outputs in response to each test case. Gulwani et al. [**?**] uses the same variable value tracing method to cluster solutions, augmenting it with information about control flow structure to overcome syntactic differences between solutions.

Statistical natural language processing techniques can also be applied to code, preferably after preprocessing. For example, Biegel et al. [**?**] described how w-shingling can capture local patterns within a solution. The w-shingling of a solution is the set of all unique subsequences of w tokens it contains [**?**]. The resemblance r between two solutions

is defined as the number of unique subsequences of w tokens both solutions contain, normalized (divided) by the union of unique subsequences of w tokens contained in either solution. It other words, it is the Jaccard coefficient of the two solutions' w-shinglings. The resemblance distance is defined as $1 - r$, which obeys the triangle inequality. n-grams models are like w-shinglings except instead of capturing just the *set* of unique subsequences of a certain length, they also capture a more global feature: the relative frequencies of these unique subsequences in the entire solution or corpus. Similarly, representing Python programs as TF-IDF vectors calculated from counts of tokens, e.g., keywords, collected across the entire corpus of solutions can capture deviations from corpus trends [**?**].

CCFinder [**?**] and MOSS [**?**] ~~(Measure of Software Similarity)~~ are both pair wise similarity (clone) detectors. Like w-shingling and n-gram models, MOSS extracts all subsequences of tokens of a specified length. Unlike them, the order of these subsequences is preserved. CCFinder [**?**] is an exception to this pattern. After aggressive pre-processing, it considers all sub-strings in both solutions and looks for matches.

The structure of solutions can be represented as trees, i.e., ASTs, and graphs, e.g., data dependency and control flow graphs. Binary or numerical feature vectors can be computed from the graphs, as well as graph-to-graph metrics of similarity, for use in feedback or assessment [**?** **?**]. Recent literature uses the full AST for computing pairwise distance metrics, e.g., the tree edit distance (TED). TED is defined as the minimum cost sequence of node edits that transforms one AST into another, given some cost function.

While a naive TED algorithm scales very poorly with tree size, an optimized TED algorithm [**?**] makes this computation more feasible. The optimized algorithm is only quadratic in both the number of solutions and the size of their ASTs [**?**]. In contrast, the analysis pipeline used by both OverCode and Foobaz scales linearly with the number and size of solutions.

Huang et al. [**?**] used the optimized TED algorithm to process approximately a million solutions while executing on a computing cluster. The same analysis pipeline was used by Roger et al. [**?**]. Yin et al. [**?**] defined a normalized TED that weighs edits associated with

nodes closer to the tree root more heavily than nodes closer to the leaves. This prioritizes high-level structural similarities between solutions and de-emphasizes minor differences in syntax near the AST leaves.

Clustering Student Solutions

Clustering student solutions can be difficult. Teachers may not agree on which solutions are closest to each other [**?**]. Rogers et al. [**?**] found that official graders for a large programming course agreed on solution clusters only 47.5% of the time even when there were only 3 possible clusters to choose from. In spite of the lack of agreement across some expert code evaluators, several research efforts have focused on automatically clustering solutions.

Luxton-Reilly et al. [**?**] suggest that identifying distinct clusters of solutions can help instructors select appropriate examples of code for helping students learn, e.g., in accordance with the systematic variation suggested by variation theory. They also suggest that it is helpful for teachers' own understanding and quality of feedback and guidance. Clustering can also be used to guide rubric creation.

Luxton-Reilly et al. [**?**] develop a hierarchical clustering taxonomy for types of solution variations, from high- to low-level: structural, syntactic, or presentation-related. The structural similarity between solutions is captured by comparing their control flow graphs. If the control flow of two solutions is the same, then the syntactic variation within the blocks of code is compared by looking at the sequence of token classes. Variation in presentation, such as variable names and spacing, is only examined when two solutions are structurally and syntactically the same. However, it is not yet fully implemented.

Other systems rely on clustering algorithms applied to solutions whose pairwise distances are determined by TED scores. AutoStyle [**?**] uses the OPTICS clustering algorithm to cluster solutions based on normalized TED scores. Huang et al. [**?**] and Rogers et al. [**?**] cluster solutions by creating a graph where nodes are solutions and an edge between each

pair of nodes exists if and only if the TED between their ASTs is below a user-specified threshold. Modularity is used to infer clusters within the graph.

Gross et al. [**?**] use the Relational Neural Gas technique (RNG) to cluster graded solutions and find solutions, called prototypes, that can represent entire clusters. Feedback on new solutions is provided by highlighting the differences between the new solution and the closest prototype. Seeing these differences can help students debug their code.

Unlike the work in this thesis, Gross et al. focus on providing feedback to a single student at a time. Graded solutions are clustered, and these clusters help identify problems in new solutions. In contrast, OverCode, Foobaz, and GroverCode process ungraded solutions and help staff compose feedback for the whole class or assign grades or feedback to a whole body of solutions at one time. OverCode and GroverCode do highlight differences between solutions to help pinpoint problems, although it is staff, rather than students, who view these differences.

Inconsistent holistic grades could be explained by teachers relying on different internalized rubrics. Teachers may be more consistent if, rather than generating holistic grades, they can annotate or grade particular components, mistakes, or design choices within solutions. Three existing approaches support this goal: (1) Create a classifier for components that teachers are interested in, similar to what was done for web pages in the Webzeitgeist dataset by Lim et al. [**?**]. (2) Model solutions as mixtures of components using mixture modeling. They have already been applied to source code and student solutions to open-response mathematical questions [**?** **?**]. (3) Create a code search engine which takes AST nodes or subtrees as queries and retrieves solutions in the database that contain them, as Nguyen et al. [**?**] did for student solutions and Fast et al. [**?**] did for general open source code.

Visualizing and Interacting with Student Solutions

There are several existing visualizations or interfaces that help teachers and students understand how solutions vary within a large corpus of solutions to a common problem. A

common design choice is to map each solution to a point in some feature space. Huang et al. [**?**], Rogers et al. [**?**], AutoStyle [**?**], and ~~Ned Gulley~~Cody [**?**] all use this strategy.

Ned Gulley designed solutions maps for Cody, a Matlab programming game[1], to help users pick and compare pairs of solutions from hundreds of solutions to the same programming problem. The solution map plots each solution as a point against two axes: time of solution submission on the horizontal axis, and code size on the vertical axis, where code size is the number of nodes in the parse tree of the solution. Users can select pairs of points to see the code they correspond to side-by-side beneath the solution map. Despite the simplicity of this metric, solution maps can provide quick and valuable insight when exploring differences among large numbers of solutions [**?**]. This can help game players learn alternative, possibly better, ways to solve a problem using the Matlab programming language, including its extensive libraries.

Huang et al. [**?**] and Rogers et al. [**?**] create graphs where each node is a solution and links between nodes indicate similarity scores beneath a certain threshold. Inter-node and inter-cluster distances correspond to syntactic similarity. Clusters are colored using modularity, a measure of how well a network decomposes into modular communities. Rogers et al. [**?**] built a grading interface on top of this clustering process. Graders graded one solution at a time, grouped by cluster.

AutoStyle [**?**] visualize all solutions on the screen using a t-SNE 2D visualization. Similar to the previous clustering interfaces, each point represents a solution and its color indicates its cluster. Hovering over a point reveals the solution it represents. Using this interface, teachers hand-annotate each cluster with a label, i.e., “good”, “average”, or “weak” and a hint about how to improve the solution. For each cluster, the teacher must also choose an exemplar solution from a *slightly better* cluster as an example of what to shoot for.

Tools that are not built for representing an entire corpus, such as file comparison tools, do have useful features to consider when designing new interfaces. Most highlight inserted, deleted, and changed text. Unchanged text is often collapsed. Some of these tools are

[1] mathworks.com/matlabcentral/cody

customized for analyzing code, such as Code Compare. They are also integrated into existing integrated development environments (IDE), including IntelliJ IDEA and Eclipse. These code-specific comparison tools may match methods rather than just comparing lines. Three panes side-by-side are used to show code during three-way merges of file differences. There are tools, e.g., KDiff3, which will show the differences between four files when performing a distributed version control merge operation, but that appears to be an upper limit. These tools do not scale beyond comparing a handful of programs simultaneously.

The OverCode interface puts the code front and center, synthesizing platonic solutions that represent entire stacks of solutions, borrowing display techniques from file comparison tools, adding filtering mechanisms and interactive clustering through rewrite rules on top of the clustering done by the analysis pipeline. OverCode was also inspired by information visualization projects like WordSeer [**?** **?**] and CrowdScape [**?**]. WordSeer helps literary analysts navigate and explore texts, using query words and phrases [**?**]. CrowdScape gives users an overview of crowd-workers' performance on tasks.

More generally, several interfaces have been designed for providing grades or feedback to students at scale, and for browsing large collections in general, not just student solutions. The powergrading paradigm [**?**] enables teachers to assign grades or write feedback to many similar answers at once. Their interface focused on powergrading for short-answer questions from the U.S. Citizenship exam. After machine learning clustered answers, the frontend allowed teachers to read, grade, or provide feedback on similar answers simultaneously. When compared against a baseline interface, the teachers assigned grades to students substantially faster, gave more feedback to students, and developed a "high-level view of students' understanding and misconceptions" [**?**].

1.3 Teaching Principles

This section describes ideas and techniques from educational psychology and the learning sciences that influenced this thesis work. The systems in this thesis are designed to support

teachers and students in massive classrooms. For example, one way to support teachers is to give them a better idea of the solution space, so they can better help a student one-on-one. Another way to support teachers is to deploy personalized prompts that mimic what the teacher might have said to the student if they could interact one-on-one.

One-on-one or small group tutoring has been held up as a gold standard in education since 1984 when Bloom published a collection of his lab's work demonstrating tutoring's efficacy relative to other experimental and conventional methods at the time [**?**]. For example, his lab found that, after 11 sessions of instruction in probability or cartography, elementary and middle school students who received tutoring in groups of one to three were, on average, two standard deviations better than their counterparts in a conventional 30-person classroom. Given the expense of scaling up one-on-one tutoring, Bloom challenged the academic community to find a method of group instruction that was just as good, or better. This became known as Bloom's two sigma problem.

Foobaz and the learnersourcing workflows explicitly prompt students to reflect and generate explanations on their own. These explanations are helpful to others, but they are also helpful to the student who generated them. These explanations, sometimes called self-explanations, foster the integration of new knowledge. Effective tutors encourage self-explanation by prompting students with questions like *Why?* and *How?* [**?**]. Students of tutors who fostered self-explanations had learning gains similar to those whose tutors provided their own explanations and feedback[**?**].

Self-explanations are a form of reflection, which is a critical method for triggering the transformation from conflict and doubt into clarity and coherence [**?**]. Turns et al. [**?**] argue that the absence of reflection in traditional engineering education is a significant shortcoming. This thesis contributes systems designed, in part, to address that shortcoming.

OverCode and Foobaz are designed to help teachers give faster and more personalized feedback to massive numbers of introductory programming students. Rapid personalized feedback supports learning in foundational engineering classrooms [**?**].

Rapid feedback is also a critical part of deliberate practice, a targeted form of concentrated practice that helps build a specific skill. Deliberate practice is goal-directed, effortful, repetitive, accompanied by rapid feedback, and only sustained as long as the learner can be fully concentrated on the task, i.e., no more than a few hours [**?**]. For example, rather than just playing pickup basketball games in the neighborhood, an aspiring professional player might design specific drills to work on his/her weaknesses. Recent work ~~incorporaing~~ incorporating deliberate practice in large classrooms has demonstrated great benefits. A recent study of undergraduate physics classrooms found that, with deliberate practice as a base of the instructional design, improvements can approach and exceed Bloom's 2-sigma threshold [**?**].

Teachers help facilitate deliberate practice, because they can design appropriate exercises and provide feedback until the student can differentiate between good and bad performance and provide that feedback to themselves. Following this pattern of instruction, Foobaz introduces a new way for teachers to provide systematic personalized feedback on good and bad variable names. The student can use this feedback to improve their own solutions and build up their own mental model about variable names are good and bad.

The comparison learnersourcing workflow was designed with zones of proximal development (ZPD) and scaffolding in mind. The concept of the ZPD was first introduced in the mid-1920's by the Soviet pyschologist Lev Vygotsky. It refers to the gap between what a learner can do without help and what a learner cannot yet do, no matter how much help they are given. It is implied that an object of learning strictly outside the ZPD is either too easy or too hard, and little or no learning will occur.

The comparison workflow dictates showing students better and worse solutions, relative to the solution they generated on their own. Some solutions are optimized in ways that may make them difficult to understand for a student who struggled just to make a working, non-optimized solution. Teachers who use the comparison workflow can decide whether students are prompted to reflect on slightly better and worse solution or radically better and worse solutions. Ideally, these better and worse solutions are not so different, they are

outside a student's theoretical ~~zone of proximal development.~~ ZPD.

~~The concept of the zone of proximal development (ZPD) was first introduced in the mid-1920's by the Soviet pyschologist Lev Vygotsky. It refers to the gap between what a learner can do without help and what a learner cannot yet do, no matter how much help they are given. It is implied that an object of learning strictly outside the ZPD is either too easy or too hard, and little or no learning will occur.~~

Wood et al. [**?**] introduced a complementary process called scaffolding. Scaffolding enables a learner to solve problems or achieve goals that would ordinarily be beyond their grasp because the teacher controls the aspects of the task that are initially outside the learner's abilities. Recent work suggests that the maximum learning gains come from giving students the hardest possible tasks they are able, with the assistance of scaffolding, to complete [**?**].

Formative feedback [**?**] can be helpful as part of the scaffolding. It should non-evaluative, supportive, timely, and specific. It usually arrives as a hint, an explanation, or a verification based on the student answer. Likewise, the comparison workflow, identifies where a student solution is on the spectrum of optimality and prompts the student to reflect on a better solution, such as the next most optimal one.

1.4 Learning through Variation in Examples

The work in this thesis is deeply influenced by multiple theories about the role of variation in learning. Designing sets of examples that illuminate an object of learning can have signifiant effects on how students understand, generalize, and transfer their learning to new contexts. This section summarizes these theories and how they have been applied in studies of human learning.

Concrete examples of an object of learning–like how to apply an appropriate statistical test in a statistics word problem–vary in ways that may be superficial or fundamental. In the language of educational psychologists, these are often called surface and structural features

[**?**]. A simple compare and contrast exercise when solving equations [**?**] or examining case studies in negotiation [**?**] can bring this variation to the fore, and yield learning benefits.

Learning in the presence of variation in these features helps learners generalize and transfer their knowledge to new situations, such as better transfer of geometric problem solving skills [**?** **?**]. Several educational models, e.g., variation theory and the 4C/ID Model [**?**], build on the value of variability by suggesting specific ways for how it should be deployed in the classroom. The three components in this thesis, OverCode, Foobaz, and the targeted learnersourcing workflows, are all designed to make the natural variability present in student solutions useful to teachers and students, in accordance with recommendations from the educational theories that follow.

1.4.1 Analogical Learning

Analogies are central to human cognition. They can help learners understand and transfer knowledge and skills to new situations. Analogical learning is at play both when learners have a base of knowledge that they bring with them to a novel target and when they compare two partially understood situations that can illuminate each other [**?** **?**]. However, in order to reap the full benefits of analogical learning, learners must engage deeply. Reading two cases, serially, in a session is not enough. Learners will not necessarily make the necessary connections unless there are explicit instructions to compare [**?** **?**].

Kolodner [**?**] suggests creating software tools that align examples to facilitate analogical learning. This thesis is, in part, a response to this suggestion. The comparison learnersourcing workflow pairs solutions the student wrote themselves with solutions that are novel to them. They are prompted to compare the solutions and write a hint for future students.

Novices may become confused if asked to compare their solution to a fellow student's solution. This is not necessarily bad for learning outcomes. Piaget theorized that cognitive disequilibrium, experienced as confusion, could trigger learning due to the creation or restructuring of knowledge schema [**?**]. D'Mello et al. maintain that confusion can be

productive, as long as it is both appropriately injected and resolved [**?**]. Similarly, reflecting on a peer's conceptual development or alternative solution may bring about cognitive conflict that prompts reevaluation of the student's own beliefs and understanding [**?**]. The comparison workflow component of this thesis is designed to stimulate this kind of productive confusion, comparison, and resolution through self-explanation.

Analogical learning can be very difficult. For example, the structural features may be aligned between a base and the new target situation, but large differences in surface features will hurt the learner's ability to see any connection [**?**]. This may be explained by how human memory works. For novices, the most reliable form of retrieval is based on surface similarity, not deep analogical similarity. Experts can more easily retrieve situations that are structurally similar and therefore more relevant for a new situation at hand [**?**]. Variation theory, discussed next, is specifically designed to help students more deeply appreciate structural features, which may help them transfer their learning to new situations instead of feeling lost, confused by superficial differences.

1.4.2 Variation Theory

Variation theory (VT) [**?**] views learning and understanding as discernment, specifically of the various features of objects and concepts that define what they are. An aphorism captures the ideas at the heart of variation theory well: *He cannot, England know, who knows England only.* VT is concerned with the way in which students are taught from concrete examples. It is relatively new and still being investigated for its usefulness in a broad range of disciplines, including mathematics and computer science.

VT is built on the understanding that learning is not possible unless the learner can discern what the object of learning is [**?**]. Discernment is not possible without experiencing variation in the object of learning and the world in which it is situated [**?**]. Dimensions of variation are described by features [**?**][2]. Some feature values are irrelevant, while critical

[2]In variation theory literature, the nomenclature is similar but distinct from that of machine learning: features are referred to as aspects and feature values are referred to as features.

feature values collectively define the object of learning.

Through the lens of machine learning, VT asserts that human learning can suffer from overfitting for some of the same reasons that machine learning algorithms do. If a machine learning algorithm selects the the color of the sky as the key difference between photos of cats and dogs (which is obviously unrelated to distinguishing between photos of cats and dogs), then a possible explanation is that the algorithm was trained on photos with insufficient or biased variation. If all the cat photos it ever saw were taken on a cloudy day, and all the dog photos it ever saw were taken on a sunny day, could you blame this naive program for latching on to this obvious differentiator of housepet species? Humans can make the same inferential mistake when not exposed to a sufficient variety of examples of an object of learning. VT catalogues a hierarchy of patterns of variation designed to immunize the learner to this kind of mistake.

For a more concrete discussion of variation, consider the following examples:

1. The phrase *a heavy object* might not make sense to the reader unless they have interacted with objects of various weights.
2. Consider a child who recently learned how to add numbers, but always starts with the larger number: 2+1=3, 4+2=6, etc. Asking the child to add the numbers in the opposite order, smaller number first, and verify that the result is the same introduces the commutative feature of addition.
3. No matter how wildly a cup diverges from a prototypical example of a tea cup, if it does not have the critical feature of being able to hold something, it is not a cup.

Marton et al. [**?**] identify four patterns of variation: *contrast*, *separation*, *generalization*, and *fusion*. The contrast pattern of variation includes examples that differ by feature values that determine whether each example represents the object of learning. If a child is learning the concept of three, then contrast refers to being introduced to three apples, as well as a pair of apples, or a dozen. The generalization pattern highlights, through contrast, what about an object of learning remains constant while certain features vary. Using this pattern of variation, a child learning the concept of three might be introduced to different groups of

three, e.g., three apples, three dogs, three beaches, and three languages. This clarifies that it is not the apples that give *three* its meaning. Separation refers to a pattern of examples that helps the learner separate a dimension of variation from other dimensions of variation. A child could be introduced to a litter of nearly identical puppies that only differ in coat color, for example. Fusion is the final pattern of variation, where the learner is exposed to examples that vary along all the dimensions of variation at once, since this is most commonly encountered in the real world. These patterns of variation are intended to reveal which aspects of a concept or phenomenon are superficial and irrelevant and which are innate and critical to its definition.

VT is a framework that has guided teaching materials and been used as an analytic framework in a variety of contexts, including lessons on critical reading [**?**], vocabulary learning [**?**], the color of light [**?**], mathematics [**?**], chemical engineering [**?**], Laplace transforms [**?**], supply and demand [**?**] and computing education [**?**]. It has been the subject of a government-funded three-year longitudinal study in Hong Kong, with promising results [**?**].

In this thesis, Overcode and Foobaz are explicitly designed to discover and make human-interpretable the variation naturally present in student solutions. All the systems in this thesis demonstrate ways in which extracted variation, in solutions or errors, can be used in massive classes.

1.5 Personalized Student Support and Automated Tutoring

Human tutoring can be very effective [**?**] but expensive to scale up. Effective human tutors often have characteristics described by Lepper and Wolverton's INSPIRE model: superior domain and pedagogical content knowledge, nurturing relationships with students, progressive content delivery, Socratic styles that prompt students to explain and generalize, and feedback on solutions, not students [**?**].

Several types of solutions have been deployed to help students get the personalized attention they need, without relying solely on human tutors. These solutions span the spectrum from recruiting more teaching assistants from the ranks of previous students [**?**] to automating hints using program synthesis or intelligent tutoring systems.

Singh et al. [**?**] uses a constraint-based synthesis algorithm to find the minimal changes needed to make an incorrect Python solution functionally equivalent to a reference implementation. The changes are specified in terms of a problem-specific error model that captures the common mistakes students make on a particular problem. The system can automatically deliver hints to students about these changes at various levels of specificity.

Intelligent tutoring systems can provide personalized hints and other assistance to each student based on a pre-programmed student model. For example, previous systems sought to provide support through the use of adaptive scripts [**?**], or cues from the student's problem-solving actions [**?**]. Despite the advantage of automated support, intelligent tutoring systems often require domain experts to design and build them, making them expensive to develop [**?**]. Furthermore, domain experts who generate these hints may also suffer from the *curse of knowledge*: the difficulty experts have when trying to see something from the point of view of a novice [**?**].

Unlike intelligent tutoring systems, the HelpMeOut system [**?**] does not require a pre-programmed student model. It assists programmers during their debugging by suggesting code modifications mined from debugging performed by previous programmers. However, the suggestions lack explanations in plain language unless they are added by experts (teachers), so the limits imposed by the time, expense, and curse of knowledge of experts still apply.

Rivers and Koedinger [**?**] propose a data-driven approach to create a solution space consisting of all possible paths from the problem statement to a correct solution. To project code onto this solution space, the authors apply a set of normalizing transformations to simplify, anonymize, and order syntax. The solution space can then be used to locate the potential learning progression for a student solution and provide hints on how to correct

their attempt.

Discussion forums derive their value from the content produced by the teachers and students who use them. These systems can harness the benefits of peer learning, where students can benefit from generating and receiving help from each other. However, as the system has no student model, the information is available to all students whether or not it is ultimately relevant. Students can receive personalized attention only if they post a question and receive a response.

Peer-pairing can stand in place of staff assistance, to both reduce the load on teaching staff and give students a chance to gain ownership of material through teaching it to someone else. Weld et al. speculate about peer-pairing in MOOCs based on student competency measures [**?**], and Klemmer et al. demonstrate peer assessments' scalability to large online design-oriented classes [**?**]. Peer instruction [**?**] and peer assessment [**?**] have been integrated into many classroom activities and have also formed the basis of several online systems for peer-learning. For example, Talkabout organizes students into discussion groups based on characteristics such as gender or geographic balance [**?**].

Recent work on learnersourcing proposes that learners can collectively generate educational content for future learners while engaging in a meaningful learning experience themselves [**? ? ?**]. For example, Crowdy enables people to annotate how-to videos while simultaneously learning from the video [**?**]. The targeted learnersourcing workflows presented in this thesis expand on learnersourcing by requesting the contributions of specific learners who, by virtue of their work so far, are uniquely situated to compose a hint for fellow learners.

Other important forms of personalized support for learning include peer groups, home environment, learning communities, and identity formation [**? ?**], but they are outside the scope of this thesis.