Research Statement

Research Goals

Data science, and machine learning in particular, is an increasingly important part of scientists' and practitioners' tool kits. While data analysis shares aspects with traditional software engineering, it presents a unique set of challenges. In particular, developers who write data analysis code tend to be either domain experts with limited software engineering experience or software engineers with limited data analysis experience. My research goal is to automate the use of data science tools to enable both groups of users to produce effective data analyses.

This research goal is influenced by my own experiences at the intersection of software engineering and data analysis. Prior to graduate school, I spent three years as a mortgage analyst at Morgan Stanley. More recently, I have collaborated with physicians at Beth Israel Deaconess Medical Center, a major teaching hospital in the Boston area, to develop a prediction model for early diagnosis of pancreatic cancer [1]. In both settings, I worked with domain experts and software experts. And in both settings, I observed how a significant amount of time is spent translating domain knowledge into software implementations, repairing invalid software assumptions resulting from lack of familiarity with the underlying data and domain, and communicating methodology and results to team members with different areas of expertise. Facilitating these interactions could accelerate and increase the impact of work in fields as different as finance and healthcare.

Given the diversity of tasks encompassed by data analysis, I believe the most scalable way to contribute to this goal is by designing and implementing novel developer assistance tools, which can automate tedious or difficult data analysis and engineering tasks. To develop these tools, my research exploits code-related artifacts as a source of rich development information. Here code artifacts are broadly construed, including, for example: existing source code repositories, online question/answering forums, program executions and environments, tool documentation, and more.

Thesis overview. My thesis research focused on learning from code artifacts to enable the automatic generation of supervised machine learning pipelines. Supervised machine learning pipelines are often implemented by composing and configuring modular components from third-party libraries such as Python's Scikit-Learn. Given the large number of possible pipelines, with varying performance characteristics depending on the input dataset, automated machine learning (AutoML) systems have been developed to automatically generate and evaluate machine learning pipelines for a specific dataset. I developed a system that learns a model for operator composition in ML pipelines based on the dynamic execution traces of existing user programs. I then developed a system that allows users to extend an initial set of API components to a full search space by automatically incorporating additional API components and constructing hyperparameter search spaces. Finally, I developed a system that frames improving machine learning pipelines' predictive performance as a program repair task and learns repair rules from a corpus of existing pipelines.
These systems aim to address three key shortcomings of existing automated machine learning approaches. AutoML systems' search spaces are defined a priori by the system developers, which

José Cambronero
jcamsan@mit.edu • *www.josecambronero.com*

may have failed to identify all necessary components for a user's dataset. A user's ability to influence the AutoML search procedure is limited to providing the input dataset, possibly resulting in pipelines that may not be desirable. And finally, AutoML tools do not currently help a user who already has a functional pipeline and is interested in improving its performance as a result of a small number of changes.
In the following sections, I will describe each of these projects in more detail, followed by future directions I am particularly excited about pursuing.

AL: Learning from dynamic traces

AutoML systems are typically written to produce pipelines using components drawn from a particular machine learning library. The portion of the library used in pipelines is typically determined by the original system developers, who identified components that work well with their search procedure. These search procedures can vary from random search to genetic programming to Bayesian optimization. In contrast, **we developed the first system, AL [2], to generate machine learning pipelines by learning from program executions.** In particular, AL identifies the subset of the library to use based on these programs, and then learns to sequentially compose operators by modeling the likelihood of adding a particular operator to a partial pipeline.

Approach. AL collects a set of approximately 500 executable programs from the public data science competition website Kaggle. AL instruments the programs to collect calls to the target Scikit-Learn and XGBoost libraries. This instrumentation provides dependency information for call parameters and return values. Parameters are further summarized using a set of meta-features, such as the types of columns or the number of rows. Based on the call dependency information, AL extracts a portion of the trace that consists of calls to components that transform the input dataset or apply a learning algorithm. Using these portions of the traces, AL trains two separate regularized logistic regression models that predict the next component in a pipeline given the previous component and the (meta-feature-based) state of the input dataset. One of the models is trained to predict the next transformation component, while the other predicts the learning algorithm at the end of a pipeline. AL then uses these two models to rank transform and learning components for a partial pipeline. AL generates new pipelines for a user by performing beam search up to a given depth bound, pruning unlikely pipeline prefixes, and returning a set of pipelines sorted by performance on a validation split of the input dataset.
Comparing AL to two existing AutoML tools, we find that AL can produce pipelines of comparable predictive performance despite the use of default hyperparameter values for all library components. Furthermore, AL can successfully produce pipelines for datasets with varied datatypes that existing AutoML systems fail to execute on. These failure stem from a lack of necessary transform components in their a priori defined search spaces or assertions in the systems' implementations that rule out possible executions as invalid.

AMS: Mining code and API documentation

The standard AutoML paradigm assumes that the user treats the search procedure as a black box: the user provides an input dataset and obtains one (or more) pipelines, but does not provide any additional feedback or information. While some tools allow users to influence the search by defining a subset of the target library and the hyperparameter space to search, effectively doing this requires machine learning knowledge. Increasing the required knowledge negates a key benefit of AutoML: users need not have extensive knowledge of the different algorithms available. **We developed the**

first system, AMS [3], to automatically generate an AutoML search space by extending a small set of relevant API components identified by the user.

Approach. Users provide AMS with a set of components that they believe would be useful for their pipeline. AMS automatically extends this set to include additional components and hyperparameter search spaces for each component. In particular, AMS mines the target library's API documentation to identify components with a high-degree of relatedness to the user's specified components. AMS defines relatedness using an information retrieval metric (BM25 [4]), which computes term overlap while normalizing for term frequency throughout the corpus. AMS then extends the input set of components to include library components that may work well jointly with those the user specified. To identify these components, AMS computes a normalized variant of point-wise mutual information (NPMI [5]) over all pairs of library components observed co-occurring in import statements in a large code corpus. AMS derives association rules based on these NPMI values. AMS then constructs hyperparameter search spaces by collecting the most frequently tuned hyperparameters and possible values from component constructor calls in the same code corpus. Finally, AMS composes the space configuration so far with a user chosen search procedure to produce a full search space.
AMS can produce search spaces that result in better performing pipelines, compared to alternative approaches that start with the user's initial component set and add increasing levels of sophistication. Furthermore, the distribution of components in pipelines sampled from AMS' spaces reflects the influence of the user's original input set of components.

Janus: Learning repair rules from AutoML search traces

The classical AutoML paradigm assumes the user does not already have a functioning pipeline, instead the search generates a pipeline starting from an input dataset. However, this approach wastes any development time invested by a user who already has an initial pipeline. Moreover, even if we warm-start the AutoML search with the user's pipeline, there is no guarantee that the final pipeline bears any relation to the original pipeline. We recognize that this scenario is similar to producing program patches, where a developer is more likely to focus on patches that resemble the original program [6]. In this analogous setting, the existing machine learning pipeline needs to be *repaired* to produce a better performing version, and we would like the patch to make a small number of changes. By framing the problem as program repair, we develop an approach similar to that used in traditional automated program repair [7] and specialize it to machine learning pipelines. **We developed Janus [8], the first tool to mine pipeline repair rules from a collection of existing machine learning pipelines produced during an AutoML search**.

Approach. Janus exploits the fact that most AutoML tools generate and evaluate a large number of pipelines during their search procedure. Janus collects these otherwise discarded pipelines and mines them for repair rules. Janus maps each pipeline to a tree representation, and defines pipeline distance to be the corresponding tree edit distance. To build a corpus of pipeline pairs, Janus identifies pipelines within a a given distance bound and which have a predictive performance differential on the same dataset. To mitigate the cost of computing tree edit distances over all pairs of pipelines, Janus employs a pruning approximation. This approximation corresponds to the euclidean distance over two vectors, where each element in a vector is the observed frequency of a particular token in the string representation of the corresponding tree. After constructing a corpus of paired trees, Janus extracts tree edit operations. We lift these edit rules to a typed variant called local structural rules (LSR). LSRs are typed to reflect ML pipeline semantics and have pre- and post-conditions based on the nodes where they were applied. To perform a repair, Janus ranks candidate LSRs based

on the joint probability of applying a given rule and the target application node. Janus greedily returns the first transformation that produces a tree that can be compiled and executed without exceptions on a small sample of the user's dataset.
Janus can successfully repair (i.e., improve the predictive performance of) more pipelines than a baseline approach based on random mutations. When both approaches improve a pipeline, Janus' repairs have comparable performance increases. Importantly, the repairs produced by Janus (by design) are closer to the original input pipelines than those produced by random mutations.

Future Directions

As data science continues to grow in importance, the number of new practitioners and available tools — and with them the importance of effective developer assistance — is set to grow. I want to continue to develop the techniques needed to power developer assistance tools for users at the intersection of software engineering and data analysis.
I believe that to deliver value to developers of increasingly complex analysis pipelines, techniques will need to be adapted to integrate the availability of multi-modal code-related data such as library documentation, software and data science question/answering forums, source code, and analysis outputs such as tables, graphics, and reports.
The following three projects propose to make use of one or more of these source of information to tackle open problems in the field of data analysis tools.

Testing in data science. As data science becomes embedded in scientific and software infrastructure, errors arising from improper development practices can have significant ramifications. In traditional software development tools such as unit tests, runtime checks, continuous integration/deployment platforms, and formal-methods-based tools mitigate the occurrence of errors. We can improve the quality of existing data analyses by automatically integrating similar practices. Exploiting the existence of public source code, we can characterize analysts' use of testing and assertions in existing code, mine repeated and adaptable patterns, instantiate these for a new user's dataset, and automatically suggest these tests during development.

Retargeting machine learning pipelines. Data scientists often implement prototype pipelines in computational notebooks using libraries that enable fast experimentation. However, often these same tools do not easily integrate with or scale to production systems. A significant development cost is thus incurred when engineers must manually translate and re-implement the prototype in a production-quality environment. We propose to automate as much as possible of this re-implementation effort by exploiting API documentation, source code examples, and input-output examples to perform API-based pipeline translations.

Pipelines from natural language. Scientific papers, particularly in domains such as clinical machine learning (e.g., [1]), often provide detailed descriptions of their analysis pipeline and increasingly include references to the source code repository of the underlying implementation. The increasing availability of this parallel corpus makes the prospect of generating partial or complete API-based machine learning pipelines from natural language appealing. Automatically generating such pipelines would enable practitioners to quickly implement baseline methods from existing literature and reduce the knowledge barrier for domain experts who have limited machine learning development experience.

References

L. Appelbaum, **Cambronero, José P**, J. P. Stevens, S. Horng, K. Pollick, G. Silva, S. Haneuse, G. Piatkowski, N. Benhaga, S. Duey, *et al.*, "Development and validation of a pancreatic cancer risk model for the general population using electronic health records: An observational study," *European Journal of Cancer*, vol. 143, pp. 19–30, 2021.

Cambronero, José P and M. C. Rinard, "Al: autogenerating supervised learning programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

Cambronero, José P, J. Cito, and M. C. Rinard, "Ams: generating automl search spaces from weak specifications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 763–774, 2020.

S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

G. Bouma, "Normalized (pointwise) mutual information in collocation extraction," *Proceedings of GSCL*, pp. 31–40, 2009.

Cambronero, José Pablo, J. Shen, J. Cito, E. Glassman, and M. Rinard, "Characterizing developer use of automatically generated patches," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 181–185, IEEE, 2019.

F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 727–739, 2017.

Cambronero, José P, J. Cito, M. J. Smith, and M. Rinard, "Janus: Learning repair rules for machine learning pipelines from automl search traces." `https://github.com/josepablocam/janus-public/tree/main/janus`, 2021.