Abstract

Traditional machine learning based intelligent systems assist users by learning patterns in data and making recommendations. However, these systems are limited in that the user has little means of understanding the rationale behind the systems suggestions, communicating their own understanding of patterns, or correcting system behavior. In this project, we outline a model for intelligent software based on a human computer feedback loop. The Machine Learning (ML) systems recommendations are reviewed by the user, and in turn, this information shapes the systems decision making. Our model was applied to developing an HTML editor that integrates ML with user interaction to ascertain structural relationships between HTML document features and apply them for code completion. The editor utilizes the ID3 algorithm to build decision trees, sequences of rules for predicting code the user will type. The editor displays the decision trees rules in the Interactive Rules Interface System (IRIS), which allows developers to prioritize, modify, or delete them. These interactions alter the data processed by ID3, providing the developer some control over the autocomplete system. Validation indicates that, absent user interaction, the ML model is able to predict tags with 78.4 percent accuracy, attributes with 62.9 percent accuracy, and values with 12.8 percent accuracy. Based off of the results of the user study, user interaction with the rules interface corrects feature relationships missed or mistaken by the automated process, enhancing autocomplete accuracy and developer productivity. Additionally, interaction is proven to help developers work with greater awareness of code patterns. Our research demonstrates the viability of a software integration of machine intelligence with human feedback.
Introduction

When editing a document, users generate constraints whenever they make a decision about how they wish it to look. A user editing their homepage might decide that all of the button elements in the main part of the document should be the visual property.

Information about a document’s look and feel is often represented explicitly through styles, such as style features in word processors or features like cascading style sheets (CSS) in HTML documents. By making them explicit, users can ensure that documents are entirely consistent with the rules by combining an initial document with a set of styling rules to generate a final document [7, 8]. Beginning with systems such as Ivan Sutherland’s Sketchpad, a variety of systems have enabled users to specify documents entirely through constraints, varying the constraints to generate new documents.

However, constraints are often unstable. Whenever the constraints change, the document may change in unexpected ways [9, 11].

In our approach, the system first learns patterns from examples of document elements, building a decision tree to predict the elements, element attributes, and element values given their context. Using these patterns, the system is able to offer the users predictions, surfacing predicted elements, element values, and get patterns through autocomplete while retrieving inspection to see why a specific example is as it is.
Autocomplete is a feature that prompts users with suggested completions for text the user has begun writing, intelligently predicting the most likely or useful words, phrases, or strings with respect to the current context [2, 6]. Although originally developed to aid those with physical disabilities, autocomplete has been applied to a broad range of technologies, most significantly for search engine queries and mobile keyboard word prediction.

One emerging application of context-based autocomplete is code completion, which brings with it several unique benefits as well as challenges. Developers often face time constraints when working on software, as well as difficulty with remembering and applying commonalities across different portions of code [1, 3, 4]. Integrated development environments address these obstacles through the suggestion of code expressions, statements, parameters, or functions [14, 15]. However, code completion may be unhelpful or even counterproductive when predictions are flawed, an inconvenience made all the more likely by the intricate structural patterns present in source code [5, 10]. Thus, intelligent assessments of context and the developers intent are of great importance.

In this paper, we outline an autocomplete function which utilizes machine learning algorithms to identify patterns between significant features in the HTML document. The function is integrated into a development environment, which displays the prediction and the features taken into account to make the prediction as a rule. In addition, this environment allows to user to prioritize, edit, and delete rules, making this tool predict more suitable for the programming task taking place.
Methods

In order to prompt the user with code completion options, the system follows the following steps: It tokenizes the source code to get the prediction type, gets the Abstract Syntax Tree (AST) representation of the source code and uses it to extract the relevant information in a table that is later used to train the decision tree and make predictions.

In order to fill the autocomplete with code completion options, the first step is to know the type of prediction. Each time the user changes something in the source code, the first thing our system does is to tokenize the entire source code. The tokenizer classifies each element in the source code to get a better understanding of the current situation the user is at. Using this and the current position of the cursor, the system determines the type of prediction to be performed, if any. The three prediction types are tag name, attribute name and attribute value, also referred to as tag, attribute, and value respectively (Table 1). By knowing the prediction type, the system knows the type of autocomplete options the user is expecting next.

| HTML Element | <p id = "blue" type = "red">Hello World!</p> |
|--------------|---------------------------------------------|
| Tag          | p                                           |
| Attributes   | id, type                                    |
| Values       | blue, red                                   |

*Table 1: Tag, Attribute, and Value Categories*

In order to get all the necessary information easily from the code and make a prediction, the system generates the Abstract Syntax Tree (AST) representation of the source code by
parsing it [16]. The system also highlights the HTML element the user is currently working on and saves the information from that element.

![Abstract Syntax Tree Example Visualization](image)

**Figure 1: Abstract Syntax Tree Example Visualization**

Before using the decision tree to make a prediction for any prediction type, the system needs to first train the Decision tree using what we call a training set. The training set is an array in which each entry is an object with the relevant features and the target feature. For each prediction type there are different features to take into account:

- **Tag prediction:** Parent tag and parents first attribute/value pair. The target feature is the tag name.
- **Attribute prediction:** Parent tag, parents first attribute/value pair, and current tag name. The target feature is the attribute name.
- **Value prediction:** Parent tag, parents first attribute/value pair, current tag name, and current attribute name. The target feature is the attribute value.

The system fills the training table in two ways. Either using the prioritized rules array or using the AST representation of the source code. We will explain the different between the two
at a later section in the paper. If the system uses the AST representation, it recursively traverses
the tree and adds an entry to the array each time with all the necessary features.

Figure 2: Tokenizing, Parsing, and Relationalization Interaction

Once all of the information is extracted for the current scenario, the information is added
to a feature table, which will be utilized to create the decision tree. As discussed earlier, there are
3 different feature tables: one for each prediction scenario. Below are examples of the three
different feature tables.
A Decision Tree is a tree-based graphical representation of decisions and outcomes for each decision. One of the most prominent Decision Tree algorithms is the ID3, or Iterative Dichotomiser 3 algorithm, invented by Robert Quinlan. The ID3 algorithm, typically used in machine learning and natural language processing projects, builds a decision tree from a fixed set of examples, and the resulting tree is used to classify future examples. Our project utilizes the ID3 Algorithm to create a decision tree given the abstract syntax tree discussed in the previous section.

Figure 3: Machine Learning Prediction System

We designed a demonstration for an editor using HTML that is broken up into 3 main parts. The first is an HTML Environment for the developer to code/use our intelligent autocomplete, the second is a box that shows the output of the programmer code, and the third is the user-interactive display that includes our decision tree and allows the programmer to change the decision tree based off of user preferences.
In addition to the conventional editor-display dichotomy, the right half of the editor includes a subsection that allows the user to see how a prediction in the autocomplete was made, and make amends to this rules to make future predictions.

This UI displays the current prediction at the top followed by a small main menu with three options for the user: Edit the current prediction, look at existing rules, and create a brand new rule. For the last two options, the user can select the type of rule(s) to edit.

**Figure 4: Developer-Machine Learning Interaction through IRIS**

Amend the current prediction: In this section the user is shown the features used to make a prediction and the top prediction made by the program. Besides going back to the main menu, the user has three options for the current prediction shown. The user can chose to delete this rule, meaning that in the future this prediction will not be shown for those specific features in the autocomplete. The user can also make the current rule a priority, meaning that this rule will be used to make a prediction in the future regardless of the changes made on the document the user
is working on. Lastly, the user can also change the prediction that is to be shown for those features. When the user changes the value of a prediction, this rule is automatically prioritized as well.

Look at current rules: In this section two rule tables will be displayed for the specified target feature. The first tables shows all the rules that have been prioritized by the user, either by adding a new rule or by changing a rule. The second table shows all the rules that are derived from the existing HTML document. This table does not contains the rules from the document that contradict the prioritized ones. Due to the lack of uniformity on the document, some rules in the second table might contradict each other. These two tables can be seen as the training sets used to train the Decision Trees.

Add a brand new rule: Depending the specified prediction type, multiple textboxes will be displayed for the user to type the desired feature values and prediction for the new rule. If the rule entered by the user is invalid (some important features are not specified, the rule has already been blacklisted, or the rule contradict with any of the prioritized rules), the rule will not be added and a message will be shown to the user as a notice. Each rule added by the user is automatically prioritized.

**Results/Discussion**

We conducted two studies to evaluate our approach. We first applied our approach to a corpus of HTML documents to understand the performance of the model and recommendation algorithm. Second, we conducted a user study, where participants used our system to author an HTML document.

**Recommendation Performance**
The validation for our autocomplete contains several steps. First, we use a script to retrieve 100 websites and take the html source code. Next, we download each of these html files into a folder, from which we upload all 100 files into our editor. As each runs, the validation.js program tests each prediction with the correct value by storing the correct tag, attribute, or value, removing the correct tag, attribute, or value, testing what the program predicts, and checking how close the programs prediction is to the correct element characteristic. At the end of each epoch, or training time, four values of accuracy are returned. We will further discuss the meanings of each accuracy in the next section.

To retrieve results for our validation, we ran 100 websites and applied the experiment protocol expressed above. For each website, we ran tests on tag prediction, attribute prediction, and value prediction. Each of these 3 tests included 4 metrics: Percentage in which the first prediction is correct including cases for all predictions, percentage in which the first prediction is correct only including cases for which predictions can be made, percentage in which one of the predictions is correct including cases for all predictions, and percentage in which one of the predictions is correct only including cases for which predictions can be made. Once we obtained each of these percentages, we took the metric that represents the percentage in which the first prediction is correct only including cases for which predictions can be made, and graphed these for each of the three tests: tag, attribute, and value prediction. Below are the graphs for Tag, Attribute, and Value Prediction for the prediction made being in the Top 1, Top 3, and Top 5.
User Study

Creation

Participants using IRIS finished building the outlined webpage after an average of 40.3 minutes versus 34.4 for the control group. The Welch's t-test difference in time taken was not significant, as $p = 0.06$.

The participants' code was scored using a 17-point rubric, with one point awarded toward the fulfillment of a general task specification (e.g. “Create a two-column table in the center of the page”). Participants using IRIS produced significantly higher-scoring code ($p < 0.01$), with scores of 14.9 points versus 11.4 points on average.

Several participants in the experimental group did not interact directly with the IRIS panel for the creation task. Instead, they opted to focus primarily on coding, while periodically browsing the "All Code Patterns" list to assess progress. As one participant explained, "I didn't really need to highlight samples of code I just wrote. But just having the list there helped me keep track of what work I've already done and what I have left." Others shared similar views,
noting that seeing the existing relationships in their code was useful for both evaluating task-compliance and conceiving ideas for what to develop next.

Of the participants who interacted with IRIS, their interactions largely consisted of defining new patterns. One participant explained that adding their own patterns ahead of time helped with "sticking to a plan", while another observed that it "made the autocomplete predictions more useful" by transferring preconceived relationships from the user to the system. Additionally, these participants made extensive use of pattern upvoting/downvoting, enabling them to redirect the system's attention from overlearned relationships towards user-preferred ones. These participants' code documents reflected a consistent, well-structured use of relationships throughout, suggesting that for creative work, substantial user control over pattern learning was conducive to the development of highly-organized code.

Continuation

For the continuation task, IRIS helped participants to more quickly identify and reapply pre-existing code relationships compared to the control group. Participants using IRIS often relied on autocomplete recommendations to develop code immediately and later, briefly review the "Current Code Pattern" and highlighted usage examples to double check the recommendations' applicability. With greater insight into the basis for the autocomplete recommendations, IRIS participants developed stronger trust in the recommendations and relied on them more heavily over time. In contrast, several control group participants appeared skeptical of the recommendations, either ignoring them or spending substantial time hunting for relevant usage examples by which to verify them. Furthermore, a few IRIS participants used the pattern voting feature to provide instant feedback on the recommendations. One participant
observed that "The autocomplete got even more accurate as I voted on the patterns", making
document development "easier and easier". Overall, participants in the experimental group
completed work on the webpage significantly faster (p = 0.03) than those in the control group,
finishing after an average of 57.1 minutes versus 44.1 minutes respectively.

IRIS helped participants with meeting continuation task requirements more often and in a
manner more consistent with the preexisting code. A 21-point rubric was used for assessing the
participants' code for the continuation task. Points were awarded on the basis of efficacy
(successfully fulfilling specifications) as well as consistency (applying appropriate features or
relationships). On average, the experimental group significantly outperformed (p < 0.01) the
control group at producing quality code, attaining 17.0 points versus 11.4 points respectively. A
substantial factor behind this improvement is the patterns list feature: IRIS participants depended
on the feature for learning of existing relationships, locating them by highlighting usage
examples, and reapplying them in new code. In particular, participants used the list feature to
explore patterns for HTML values, enabling them to reproduce appropriate attribute-value pairs
for a given element. Additionally, many participants browsed the HTML tag patterns list for its
parent tag conditions, which aided them at reproducing appropriate parent-child structures. For
instance, participants first became aware of the rule to nest self-contained <img> elements inside
<figure>}parents by seeing this relationship expressed in the patterns list (as a new semantic
element introduced in HTML5, the <figure> tag and its usage may not be widely understood).
Most participants followed up by highlighting code examples of this rule, and either recreating or
copy-pasting relevant portions of code involving <figure> and <img>. On the other hand, control
group participants often developed `<img>` elements without `<figure>` parents, and more generally, were not as aware of preexisting relationships in the code document.

**Correction**

Participants' code was scored using a highly-specific 27-point rubric, which awarded points for the addition, modification, or deletion of code to make it consistent with relationships used elsewhere in the document (e.g. Correct `<p>` tag on line 34 to `<caption>`). IRIS users earned 23.5 points, on average, versus 14.8 for the control group. The primary source of this significant improvement at correcting inconsistencies (p < 0.01) was the ability of IRIS participants to highlight usage examples: all participants in the experimental group employed the highlight feature to scan the document for code in noncompliance with patterns. IRIS Participants generally interpreted the red highlights to indicate a defective code feature, and the yellow highlights to suggest a missing or defective attribute-value pair. This sometimes misled a few participants into "fixing" elements that did not need correction, as certain code defects featured predominantly in the document, and thus, were listed by IRIS as code patterns. Despite this occasional shortcoming, the patterns list and highlighting features was instrumental in participants successfully finding and repairing code defects.

One IRIS Participant stated, "The magnifying glass [button] was very handy. For each pattern I pressed [the button], read the colored lines and compared them... And then figured out which [line] needed fixing from there."

Meanwhile, control group participants struggled to locate inconsistencies in the document, particularly those concerning HTML values. A control participant stated "Sorting
through all the code was really confusing and time-consuming. I couldn't tell what I was supposed to do for the most part... Only a few blatantly wrong tags stood out to me."

Owing largely to the speed granted by the highlighting feature, IRIS participants realized a significant boost in efficiency (p < 0.01) compared to their control group counterparts, finishing corrective work after an average of 29.4 minutes versus 44.1 respectively.

| Task   | Control | IRIS |
|--------|---------|------|
| Creation | 40.3    | 34.4 |
| Continuation | 57.1    | 44.1 |
| Correction | 44.1    | 29.4 |

*Figure 5: Average Minutes Taken Comparison between Control and IRIS Participants*

| Task   | Control | IRIS | Total Criteria |
|--------|---------|------|----------------|
| Creation | 11.4    | 14.9 | 17             |
| Continuation | 11.4    | 17.0 | 21             |
| Correction | 14.8    | 23.5 | 27             |

*Figure 6: Average Score Comparison between Control and IRIS Participants*

Machine learning traditionally assumes that there is little noise in the data or, if there is noise, the noise is independent of the model [11, 12, 13]. Even in cases where a user is providing the labels based on instructions from the learner on what would be most insightful (active learning), the assumption is that the user is right. But in seeing patterns, as revealed from models
and rules learned from the data, the user may change their view on what the correct data is in order to result in a simpler model. This relies on having a setting where the data is generated entirely by the user and there is no other external source of truth on what the correct data is.

**Conclusion**

We presented an approach of creating a user-interactive decision tree that has proven, in our user study, to be more effective than the standardized autocomplete commonly used by Software Engineers. Our process includes retrieving the html file, tokenizing the html file, creating the abstract syntax tree, and extracting certain features to preserve relationships and create the feature table. The feature table is then used as an input in the ID3 algorithm to produce the decision tree for the user. The created decision tree is then shown to the user as a set of revisable rules with options to add new rules or fix certain parts of each rule. Throughout this process of the user editing the guidelines for the autocomplete predictions, the users changes are reflected as the user continues to type more code, with the users specific preferences being shown before other predictions. This stylistic decision tree, therefore, helps the user program more comfortably and clarifies how each autocomplete decision is being made.

Aside from the benefit that our User-Interactive Decision Tree Autocomplete presents, the validation shows that the decisions made by our process almost always meets, if not exceeds, the accuracy, precision and recall compared to standardized autocompletes. After extensive testing on over 20 websites, we have recorded an average accuracy of 89% , which exceeds the current standard autocomplete accuracy of 75%.
In the future, we hope to look at other features for making predictions, including other attribute/value pairs for same element if any, grandparents information and decisions based off of number of attribute/value pairs.

We believe this work represents an important advance in creating powerful user-interactive models of decision trees which will be useful to developers worldwide for a variety of applications.
References

[1] Raychev, V., Bielik, P., & Vechev, M. (2016, October). Probabilistic model for code with decision trees. In ACM SIGPLAN Notices (Vol. 51, No. 10, pp. 731-747). ACM.

[2] Tam, C., & Wells, D. (2009). Evaluating the benefits of displaying word prediction lists on a personal digital assistant at the keyboard level. Assistive technology, 21(3), 105-114.

[3] Kotsiantis, S. B. (2013). Decision trees: a recent overview. Artificial Intelligence Review, 39(4), 261-283.

[4] Horvitz, E. (1999, May). Principles of mixed-initiative user interfaces. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems (pp. 159-166). ACM.

[5] Gunning, D. (2017). Explainable artificial intelligence (xai). Defense Advanced Research Projects Agency (DARPA), nd Web, 2.

[6] Kulesza, T., Burnett, M., Wong, W. K., & Stumpf, S. (2015, March). Principles of explanatory debugging to personalize interactive machine learning. In Proceedings of the 20th international conference on intelligent user interfaces (pp. 126-137). ACM.

[7] Ma, L., Destercke, S., & Wang, Y. (2016). Online active learning of decision trees with evidential data. Pattern Recognition, 52, 33-45.

[8] Wierzynski, C. (2018). The challenges and opportunities of explainable ai. Intel. com, 12.

[9] Kuang, C. (2017). Can AI be taught to explain itself. The New York Times, 21.

[10] Gunning, D. (2017). Explainable artificial intelligence (xai). Defense Advanced Research Projects Agency (DARPA), nd Web, 2.

[11] Chandrasekaran, B., Tanner, M. C., & Josephson, J. R. (1989). Explaining control strategies in problem solving. IEEE Intelligent Systems, (1), 9-15.
[12] Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., & Carey, T. (1994). Human-computer interaction. Addison-Wesley Longman Ltd..

[13] Samek, W., Wiegand, T., & Müller, K. R. (2017). Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. arXiv preprint arXiv:1708.08296.

[14] Adadi, A., & Berrada, M. (2018). Peeking inside the black-box: A survey on Explainable Artificial Intelligence (XAI). IEEE Access, 6, 52138-52160.

[15] Miller, T. (2018). Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence.

[16] Li, Y., Katsipoulakis, N. R., Chandramouli, B., Goldstein, J., & Kossmann, D. (2017). Mison: a fast JSON parser for data analytics. Proceedings of the VLDB Endowment, 10(10), 1118-1129.