2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)

Jigsaw: Large Language Models meet Program Synthesis

Naman Jain
t-namanjain@microsoft.com
Microsoft Research
Bangalore, India

Skanda Vaidyanath*
svaidyan@stanford.edu
Stanford University
Stanford, USA

Arun Iyer
ariy@microsoft.com
Microsoft Research
Bangalore, India

Nagarajan Natarajan
nagarajn@microsoft.com
Microsoft Research
Bangalore, India

Suresh Parthasarathy
supartha@microsoft.com
Microsoft Research
Bangalore, India

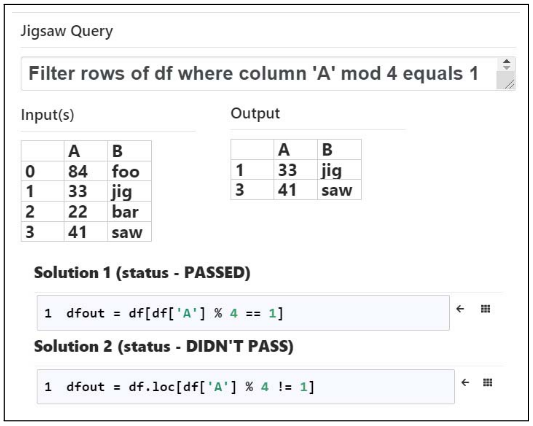
Sriram Rajamani
sriram@microsoft.com
Microsoft Research
Bangalore, India

Rahul Sharma
rahsha@microsoft.com
Microsoft Research
Bangalore, India

ABSTRACT

Large pre-trained language models such as GPT-3 [10], Codex [11], and Google's language model [7] are now capable of generating code from natural language specifications of programmer intent. We view these developments with a mixture of optimism and caution. On the optimistic side, such large language models have the potential to improve productivity by providing an automated AI pair programmer for every programmer in the world. On the cautionary side, since these large language models do not understand program semantics, they offer no guarantees about quality of the suggested code. In this paper, we present an approach to augment these large language models with post-processing steps based on program analysis and synthesis techniques, that understand the syntax and semantics of programs. Further, we show that such techniques can make use of user feedback and improve with usage. We present our experiences from building and evaluating such a tool Jigsaw, targeted at synthesizing code for using Python Pandas API using multi-modal inputs. Our experience suggests that as these large language models evolve for synthesizing code from intent, Jigsaw has an important role to play in improving the accuracy of the systems.

ACM Reference Format:
Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510203



Filter rows of df where column 'A' mod 4 equals 1

	A	B
0	84	foo
1	33	jig
2	22	bar
3	41	saw

	A	B
1	33	jig
3	41	saw

Solution 1 (status - PASSED)

```
dfout = df[df['A'] % 4 == 1]
```

Solution 2 (status - DIDN'T PASS)

```
dfout = df.loc[df['A'] % 4 != 1]
```

Figure 1: Multi-modal problem specification in Jigsaw

1 INTRODUCTION

Pre-trained large language models (PTLM) such as GPT-3 [10] are finding pervasive applications in Natural Language Processing (NLP), as a general purpose platform to solve many NLP tasks. Recent efforts show that PTLMs can generate code from natural language prompts, by associating documentation text with code from a large training set [1, 7, 11]. This presents a new avenue for program synthesis. However, PTLMs do not "understand" either the syntax or semantics of the code, and treat code as text [7]. Consequently, the code produced by such models has no guarantees of correctness or quality. Hence, any system that uses such PTLMs to generate code will need to augment it with program analysis and program synthesis modules to ensure correctness. In this paper, we present the design and empirical evaluation of such a multimodal program synthesis system called **Jigsaw**, which is targeted specifically at synthesizing code for using large and complex APIs.

Jigsaw is multi-modal (as depicted in Figure 1) in the sense that it can ingest input as (1) a natural language string expressing intent and (2) a set of test cases, or input-output examples, and produces a code snippet as output. Future incarnations may be designed to accept other modes of input as well. The architecture of Jigsaw is shown in Figure 2. The pre-processing module converts the natural language intent into a customized query to send to the PTLM. The post-processing module performs syntactic and semantic checks, and performs transformations on the code produced by the PTLM, ensuring that the code passes the supplied test cases and other quality checks. The transformations are specifically designed to correct common and recurring errors made by PTLMs, such as referencing errors (where the code references variable names incorrectly), argument errors (where the code invokes the correct API, but with incorrect arguments), and a class of semantic errors (which

[*] Work done by author during internship at Microsoft Research, India

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510203

can be corrected by learning AST-to-AST transformations). Section 2 shows concrete examples of such errors, and Section 3 shows how the transformations correct such errors. Jigsaw learns from usage by incorporating user feedback into both pre-processing and post-processing modules, and learns from user engagements to improve its overall quality. Our experiments show how Jigsaw is able to learn from past usage to improve future performance.

The current version of Jigsaw is designed and evaluated to synthesize code for the Python Pandas API [27]. However, the principles behind the design of Jigsaw are general, and the design can be extended to other libraries and programming languages as well. We create a user interface for Jigsaw using a Jupyter notebook [2] extension. The extension can be invoked using a magic command, and invocation of the command creates a sidebar window with a Jigsaw card for each invocation. The Jigsaw card allows users to supply and edit inputs to the system, inspect the results and copy the desired output back into the main notebook window.

We evaluate Jigsaw in terms of the overall accuracy, as well as accuracy of the components of the pre-processing and post-processing modules, on two datasets we created: PandasEval1, created by the authors of this paper, and PandasEval2, created by 25 users during a hackathon, where participants were given points for solving Python Pandas tasks using Jigsaw. The hackathon was conducted across two sessions (details in Section 4). We used user feedback from the first session to improve the pre-processing and post-processing modules of Jigsaw, and found users were about to solve about 10% more tasks in the second session, due to learning improvements from the first session.

In Section 5, we instantiate Jigsaw with two state-of-the-art PTLMs: GPT-3 [10] and Codex [11], and present comprehensive evaluations. We show the overall improved performance of Jigsaw compared to baselines and state-of-the-art code synthesis frameworks on the two datasets, as well as gains due to learning from user feedback over time.

In summary, this paper makes the following contributions:

- We present an architecture to perform code synthesis by augmenting black-box PTLMs with program analysis and synthesis-based techniques and multi-modal specifications. We have implemented the architecture in a tool called Jigsaw. We have developed a Jupyter notebook extension that allows users to interact with the system seamlessly.
- We characterize common classes of errors made by PTLMs, namely, reference errors, argument errors, and semantic errors. Motivated by these errors, we have designed program analysis and synthesis techniques in Jigsaw to fix such errors in code produced by PTLMs. We have also designed techniques to learn from user feedback and improve with usage.
- We have created two Pandas datasets with multi-modal specifications (released for community use). Using two state-of-the-art PTLMs, we show that Jigsaw yields significantly higher accuracy compared to baselines on the two datasets.

Our hypothesis is that even as PTLMs for code improve, systems such as Jigsaw that perform pre-processing and post-processing modules will be crucial to improve user experience, and enhance the quality of the output produced. This is because PTLMs inherently do not understand the syntax or semantics of code they generate, so we expect gaps to remain between PTLM output and user expectation. Tools based on program analysis and synthesis techniques that understand the code and API syntax and semantics can address these gaps better than generic PTLMs. We discuss how to design pre-processing and post-processing modules in a general manner, so that Jigsaw can work for any language and any API.

2 JIGSAW OVERVIEW

Jigsaw is a *multi-modal*, *interactive* code synthesis system where (a) the user specifies intent via a combination of natural language description and test cases (i.e., input-output (I/O) examples); and (b) the user interacts with the system via a friendly and seamless interface integrated with the programming environment. The interactive aspect of Jigsaw is crucial for the developer to refine the possibly ambiguous intent specification as well as for the system to gather useful feedback for improving the components. In this section, we highlight some of the challenges in using general-purpose PTLMs for specific domains with example queries, which directly motivates the design of our Jigsaw code synthesis pipeline.

2.1 Jigsaw **design principles**

We treat large language models as black-box, i.e., we can only query them. This is a reasonable assumption since the premise that the expertise and means to fine-tune the models is out of reach for most users. This design choice is motivated by three reasons: (a) there is a natural barrier to access these large models, and we can get only the output of the model for a given input via some interface (e.g., REST APIs), (b) these large language models constantly evolve and get better with each generation [10, 33]; treating them as black-boxes enables plug-and-play with minimal effort, and (c) finally, domain-specific improvements to large language models (e.g. for Python programming [15], general programming [11]) are rendered complementary to our efforts rather than competitive.

We configure Jigsaw with a PTLM (GPT-3 and Codex, in this work) of choice to be used as a black-box. We focus on appropriately setting up or priming these models for a given task at hand, characterizing common failure modes of PTLMs for code synthesis, and building components that can overcome such recurring failures. We rely on both program synthesis-based techniques and multi-modal specification to design these components. Our goal is to enable synthesis of syntactically and semantically correct code snippets for a given domain and using feedback from usage to improve the system over time.

The pre-processing module of Jigsaw contextualizes the input to the black-box language model using heuristic techniques (akin to recent efforts [23, 30, 46]). A key contribution of our system is in its post-processing module:(a) speeding up the combinatorial search space of API functions and their arguments, and (b) learning and updating a set of transformation or re-write rules to be applied to the erroneous snippets output by PTLMs. The post-processing module uses I/O examples to choose the appropriate transformation.

In the rest of the paper, we instantiate Jigsaw for solving data transformation tasks with Python Pandas API, which is widely used by data scientists to process tabular data [27].

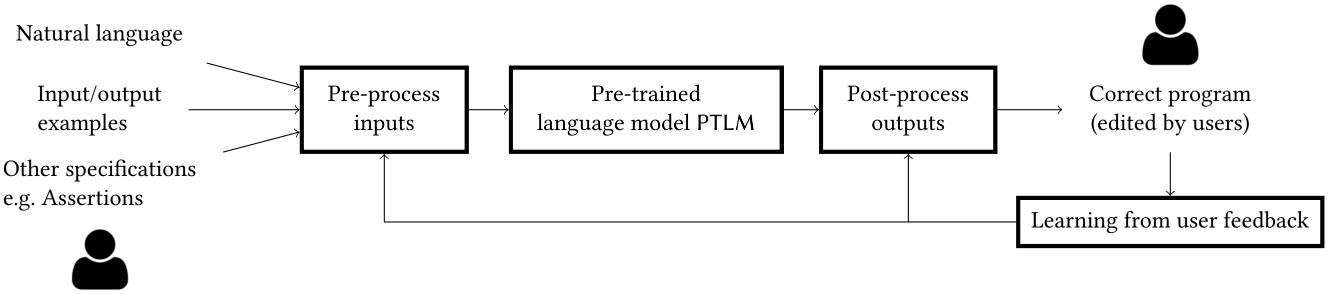

Natural language
Input/output examples
Other specifications e.g. Assertions
Pre-process inputs
Pre-trained language model PTLM
Post-process outputs
Correct program (edited by users)
Learning from user feedback

Figure 2: Architecture of Jigsaw

2.2 PTLMs as black-box

Consider a typical scenario where the user wants to load and examine the data in a csv file. Pandas uses dataframe objects (two-dimensional tabular representation) to store and process heterogeneous data (commonly named with *df* prefixes in the code, like *df, df1, df2, dfin*). The user invokes Jigsaw with a simple natural language description of the intent in a cell of Jupyter Notebook:

`%jigsaw -q "Load ./data.csv file"`

The “magic command” `%jigsaw` invokes the synthesis pipeline with the given query. Jigsaw (configured with GPT-3) returns the following snippets for the above query:

```
df = pd.read_csv('./data.csv')
csv = pd.read_csv('./data.csv', header=None)
```

The user then issues the following query in the session:

`%jigsaw -q "Remove substring ‘Name:’ from column ‘country’ of df"`

Jigsaw produces the following snippets.

```
df['country'] = df['country'].str.replace('Name:', '')
```

A key knob in PTLMs is setting the right context for a given user query. This context is passed as an input to the PTLM in addition to the user query. To this end, Jigsaw first prepares the input in the *pre-processing stage* (details in Section 3.2). The preparation involves assembling a set of “relevant” question-answer pairs to inform the PTLM of the nature of the input task — which is converting natural language text to Python code, specifically, Pandas code. With the context selection in the pre-processing stage, Jigsaw produces the desired code snippets shown above. In contrast, the GPT-3 model without context selection produces the following incorrect snippet for the above query:

```
df = df.country.str.remove('Name:')
```

Recent studies, both in the context of natural language understanding [30, 46] as well as in the programming [7] domains, have shown the influence and importance of context selection in the output of PTLMs. Our work provides further evidence that context selection can significantly impact the quality of the code generated for Pandas programming tasks, with two different PTLMs, demonstrated in Section 5.4.

2.3 Learning to fix recurring failure modes of PTLMs

The core aspect of Jigsaw system design is incorporating a post-processing phase that involves: (a) characterizing, (b) transforming the (syntactically and/or semantically) erroneous code snippets, and, more importantly, (c) endowing the system with the capability to improve (in terms of accuracy) from feedback as more users interact with it over time. Below, we highlight common classes of errors we observe over two different Pandas programming datasets (created by us, and described in Section 4) using two different PTLMs, namely GPT-3 and Codex.

1. Referencing errors: We observe that, even with suitable context, PTLMs can produce incorrect referencing of variable names in otherwise accurate code snippets.

2. Incorrect arguments: In some cases, PTLMs produce code with the right composition of API functions, but with incorrect arguments. For instance, consider the following invocation:

`%jigsaw -q "remove all duplicate entries of column 'inputB'"`

```
dfout = df.drop_duplicates(subset=['inpB']) # PTLM
dfout = df.drop_duplicates(subset=['inpB'],keep=False) # Correct
```

3. Semantic errors: A recurring failure mode for the PTLMs we have experimented with is that they produce code snippets that are *almost* correct, but the semantics are wrong because of a minor error. We can quantify this via suitable edit distance between the ASTs of the produced and the correct (i.e., intended) code snippets. For instance, consider the following invocation:

`%jigsaw -q "Get fourth value from column 'C' in dfin and assign to dfout"`

```
dfout = dfin.ix[3, 'C'] # PTLM
dfout = dfin.loc[3, 'C'] # Correct
```

Jigsaw employs a post-processing phase that critically relies on the multi-modal specification (I/O examples, in particular) to overcome the aforementioned recurring failures. To this end, we pass the incorrect output code snippet from PTLM (which can be ascertained with the help of I/O examples in the specification) through a series of components driven by PL-based techniques (details in Section 3.3). The two key ideas are outlined below.

(1) Using the API functions in the incorrect code snippets produced by PTLM, we seed the enumerative search for the right arguments. We perform this search efficiently adapting the

```
gpt3 = GPT(engine="davinci", temperature=0.5, max_tokens=100)
# Examples to train a English to French translator
gpt3.add_ex(Example('What is your name?','quel est votre nom?'))
gpt3.add_ex(Example('What are you doing?','Que faites-vois?'))
gpt3.add_ex(Example('How are you?','Comment allex-vous?'))

# Input to the model
prompt3 = "where are you?"
output3 = gpt3.submit_request(prompt3)
# Model output
output3.choices.text
```

```
Output: Où êtes-vous?
```

Figure 3: English to French translation using GPT-3

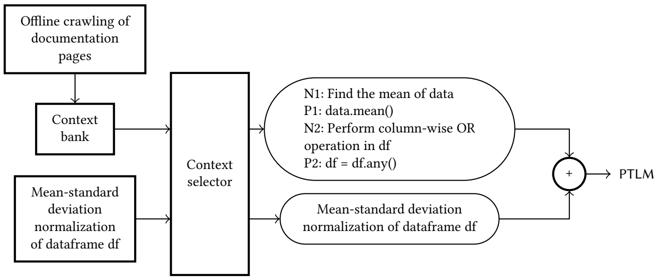

Offline crawling of documentation pages
Context bank
Mean-standard deviation normalization of dataframe df
Context selector
N1: Find the mean of data
P1: data.mean()
N2: Perform column-wise OR operation in df
P2: df = df.any()
Mean-standard deviation normalization of dataframe df
+
PTLM

Figure 4: Illustration of Pre-processing step of Jigsaw

AutoPandas framework [9] which is an enumerative-search based programming by examples framework built for Pandas API.

(2) The user interface of Jigsaw enables getting feedback which is then used by our system to learn a set of AST-to-AST transformations using the Prose synthesis framework [16, 32]. The challenge here lies in clustering errors that are *alike* so that a small set of general transformations can be learnt.

3 JIGSAW ARCHITECTURE

The architecture of Jigsaw is depicted in Figure 2. In this section, we describe each module in detail.

3.1 Pre-trained Language Models

We describe Pre-trained Language Models (PTLMs) using GPT-3 as an example. GPT-3 stands for "Generative Pre-trained Transformer 3", which is the third version of a large transformer model developed by OpenAI. GPT-3 is a neural model with 175 billion parameters, trained on a very large corpus consisting of publicly available datasets such as CommonCrawl [1], WebRText dataset, two internet-based books corpora, and English Wikipedia. GPT-3 is a general-purpose model that can be customized to perform a variety of NLP tasks. Such customizations do not involve fine-tuning the ML model for the specific task at hand. Instead, the user of GPT-3 can describe the task using a few examples (on the order of 4-5 examples works usually), and GPT-3 is then able to produce answers for the specific task. A session with GPT-3 has the form: $(Q_1, A_1), (Q_2, A_2), \ldots, (Q_k, A_k), Q$, where k is a small number (typically 4 or 5), the pairs (Q_i, A_i) are question-answer pairs to describe the task we want GPT-3 to perform, and Q is the question for which we seek an answer.

For example, if (Q_i, A_i) are such that Q_i are English statements and A_i are corresponding French translations, then GPT-3 becomes an English-French language translator. See session in Figure 3.

Other recent PTLMs include Codex [11], which is OpenAI's recent language model trained specifically on code, and Google's large language model [7]; these models translate natural language to program. Jigsaw uses PTLMs to produce Pandas code, given a natural language description of intent, and test cases. Specifically, Jigsaw session with GPT-3 has the form: $(N_1, P_1), (N_2, P_2), \ldots, (N_k, P_k), N$ where N_i is English description of intent, and P_i is the code snippet we want the PTLM to produce. We currently do not pass input-output examples to the PTLM. Instead, we use these test cases to check and filter the candidate codes produced by the PTLM during post-processing, or transform the code produced by the PTLM such that it passes the test cases.

3.2 Pre-processing

The goal of Jigsaw's pre-processing module is to convert the user intent into a suitable query for the PTLM. As mentioned above, PTLMs take a sequence of question-answer pairs (N_1, P_1), (N_2, P_2), $\ldots$, (N_k, P_k) as a preamble before we supply the current query. We use the term *context* to denote this preamble. Previous works in natural language processing (NLP) [23, 30, 46] have shown that the performance of these PTLMs is heavily influenced by this context; and the performance improves if the question-answer pairs in context are similar to the current query N. Hence, we maintain a *context bank*, of possible question-answer pairs, and then choose elements of the context bank that are similar to the current query, and add these to the context. Jigsaw creates a context bank offline by scraping and annotating examples from API documentation for Pandas, as well as other resources of examples (from tutorials, etc.) that are used to teach the API. When the user asks a question, the question is fed to a context selector that uses a text similarity metric to pick the most relevant prompts from the context bank (see Figure 4). We study two kinds of similarity metrics for the context selector: *a*) tf-idf similarity [39] (TFIDF), and *b*) transformer similarity [37] (TRANSFORMER) . The context thus produced is appended with the current query and fed as input to the PTLM.

In cases where Jigsaw is unable to produce the correct answer, we let users make changes to the incorrect Jigsaw code and use such a feedback to enhance the context bank (details in Section 3.4). PTLMs also take an input parameter called *temperature*. Lower values of temperature result in fewer accurate answers. Higher values result in a less accurate but more diverse set of answers. We report on how we pick temperature values in Section 5.

3.3 Post-processing

The code snippets produced by the PTLM vary in accuracy and quality, depending on the natural language sentences used to ask to encode the question, the context bank supplied, the context selection as well as the temperature parameter. The goal of Jigsaw's post-processing step is to filter and transform the output produced

[1] https://commoncrawl.org/the-data

by the PTLM to produce a correct answer. Our measure of correctness is that the code produced should pass the I/O examples specified by the user. In many cases, the code does not parse or fails with an exception. We consider such cases as test failures. If a non-empty set of candidate solutions produced by the PTLM satisfies the test cases, then we merely show those code snippets to the user. Our experience is that for about 30%-60% of the cases (depending on the PTLM and the dataset), the PTLM produces correct outputs. In the remaining cases, Jigsaw uses the candidate solutions produced by PTLM as starting points and performs transformations on candidate code snippets using simple program analysis and synthesis techniques to produce correct solutions. We describe the correctness checks and transformations below:

Correctness checks: In cases where we have I/O examples, we run the candidate code snippet starting with each of the specified inputs, and check if the output produced agrees with the corresponding specified output. This check can be expanded to include static analyses to check for security vulnerabilities and other errors, as motivated by recent work [29].

Variable Name transformations: In some cases, PTLMs produce accurate code snippets, but with incorrect variable names. This is often due to the model’s bias towards common variable names like `df` for Pandas dataframes and also because users assume variable referencing to be implicit. As an example, we find that GPT-3 produces the code snippet `df1.merge(df2)` when the correct answer is `df2.merge(df1)`. Since users specify inputs and output variables in the natural language description or in test cases, this post-processing step uses such information from multi-modal inputs, as well as names of variables in scope, by systematically searching over potential variables, and trying possible permutations and combinations of variable names so as to pass the test cases.

Argument transformations: In some cases, the PTLMs produce code snippets with correct method names and method sequences (in case multiple methods need to be invoked in a nested manner or one after another), but with incorrect arguments. As an example, in response to the query “`replace ‘United States’ in ‘location’ by ‘US’ and ‘3434’ in ‘zip’ by ‘4343’`”, Codex produces:

```
dfout = df.replace({'United States':'US', 3434:4343})
```

This snippet invokes the correct method `replace`, but misses the detail in the question that `‘United States’` and 3434 must be replaced with `‘US’` and 4343 *only* when these values are present in the columns `‘location’` and `‘zip’` respectively. The correct code synthesized by Jigsaw for this query is as shown below:

```
dfout = dfin.replace({'location':{'United States':'US'},
          'zip':{3434:4343}})
```

Motivated by such cases, this post-processing step systematically searches through arguments from an inferred argument space for a given sequence of method/function names. In order to implement the systematic search over the space of arguments, we adapt the approach used by Autopandas tool [9], with the following modifications. Autopandas uses a Graph Neural Network, that takes I/O examples as input, to choose method names. However, we need a lot of domain-specific data to train such neural networks. In our case, we simply extract the method names from the output of PTLM given the natural language query (which readily scales to programming domains beyond Pandas). The argument space to perform the search is inferred using the natural language text input, the arguments present in the PTLM output, the column names from the dataframe schema as well as variables in scope. We extend the generators in Autopandas to consider complex data types such as lists and dictionaries, and we extend the set of APIs considered to include APIs that return Pandas Series types (one-dimensional labeled arrays capable of holding data of any type) in addition to the ones that return Pandas dataframe types. With these modifications, we find that Jigsaw is able to transform several incorrect code snippets produced by the PTLM to correct code snippets (as shown in Section 5).

AST-to-AST transformations: In some cases, we find that the PTLM produces code that is almost correct but has a minor error. We also find that such errors are *repeatedly* made by the PTLMs. As a specific example, we find GPT-3 often misses the bitwise not operator, and produces the code:

```
train = data[data.index.isin(test.index)]}
```

instead of the following correct code with the bitwise not operator:

```
train = data[~data.index.isin(test.index)]}
```

As another example, we find that GPT-3 misses paranthesizations, which results in the generated code raising an exception. Specifically, the generated code is:

```
dfout = dfin[dfin['bar']<38|dfin['bar']>60]
```

instead of the following code synthesized by Jigsaw which is parenthesized correctly:

```
dfout = dfin[(dfin['bar']<38)|(dfin['bar']>60)]
```

Such errors cannot be fixed via variable name transformation or argument transformations. However, code has well-defined structure, usually represented as abstract syntax tree (AST). Jigsaw takes advantage of this structure and corrects such errors by learning re-writing rules as AST-to-AST transformations learned from user interactions with Jigsaw. These transformations are applications of production rules from grammar used in BluePencil [26] which is used for suggesting code re-factorings. However, it is not possible to learn these rules at the appropriate level of generality from a single example. This generality is necessary so that the missing negation or parenthesizing can be corrected by the learnt transformation, even if the same pattern is repeated with a different set of variables or constants. To achieve this, we collect data from user interactions, where the user edits the answer produced by Jigsaw to produce the correct code. We cluster the data points (i.e., code snippets) so that similar data points are grouped together and we learn a single AST-to-AST transformation that is able to handle all the data points in a cluster. Unlike the case of refactoring where users will implicitly hint at clustering of similar edits (by attempting them one after the other), we resort to a greedy heuristics-based clustering algorithm. This clustering is performed in an online fashion as we get more data points for learning AST-to-AST transformations. For each data point, we decide if the data point is grouped inside an existing cluster or instantiate a new cluster. In the former case, we check if the AST-to-AST transformations from the existing cluster can be re-learnt to be more general, and if so, re-learn the transformations. In addition, we perturb the data points in each cluster to change variable names and constants, in order to prevent learning transformations that over-fit. Together with the above-mentioned clustering and perturbation heuristics, we find that we are able to learn transformations at the appropriate level of generality (Section 5.2.1). We use the PROSE program synthesis system [16, 32] to learn the transformations from a cluster of incorrect-correct code snippets. While Jigsaw currently works only on Python code, the post-processing step works at the level of ASTs, and can be made to work across programming languages as well.

We refer to Argument transformations and AST-to-AST transformations together as Semantic Repair in experiments (Section 5).

3.4 Learning from user feedback

The user interface of Jigsaw (integrated into the Jupyter notebook) is designed to let users submit correct code in cases where Jigsaw is incorrect. Jigsaw can be improved by assimilating user feedback. Specifically, we design techniques for updating context-bank in the pre-processing module and AST-to-AST transformations in the post-processing steps, as more users interact with Jigsaw.

Updating context bank: The procedure for updating context bank with user queries is given in Algorithm 1. We first check whether Jigsaw already found a correct solution for the given (new) query N, thus giving us some confidence about its correctness. Otherwise, we check if any of the solutions generated by Jigsaw is "close" to some correct code (determined by the standard edit distance on strings $\mathbf{d}_{\mathsf{EDIT}}$ and a chosen threshold ϵ_{CODE}). If either of the two conditions is satisfied, we add the new query to our context bank while additionally ensuring that a similar query already does not exist (via TFIDF based distance $\mathbf{d}_{\mathsf{TFIDF}}$ and a threshold ϵ_{BANK}). With more usage, we grow the context bank and try to cover different styles of user queries, which in turn helps relevant context selection.

Algorithm 1 Updating context bank

```
Inputs:
    Context Bank : $\mathbf{C} = \{(N_1, P_1), (N_2, P_2), \ldots, (N_{|\mathbf{C}|}, P_{|\mathbf{C}|})\}$,
    New query and feedback (code snippet): $N, P$
Output: Updated Context bank $\mathbf{C}$
Let output = Jigsaw($N$, $\mathbf{C}$)
If $\min_i \mathbf{d}_{\mathsf{EDIT}}(\text{output}_i, P) > \epsilon_{\mathsf{CODE}}$
    return $\mathbf{C}$
If $\max_i \mathbf{d}_{\mathsf{TFIDF}}(N, N_i) < \epsilon_{\mathsf{BANK}}$ return $\mathbf{C}$
return $\mathbf{C} \cup \{(N, P)\}$
```

Updating transformations: For every query paired with correct code snippet(s), we select all incorrect codes suggested by PTLM within some small edit distance of a correct code. The AST-to-AST transformations learning sub-module performs clustering (with perturbations) on the selected code snippets as discussed in the above subsection, and updates the set of transformations.

3.5 Generality of approach

We believe that the ideas presented above such as context selection, correctness checking, and transformations are general and that it is possible to design pre-processing and post-processing steps in a generic manner that can work across languages, APIs, and PTLMs. We give some evidence to this argument below: *a*) Though we did our initial analysis on Pandas code generated by GPT-3, we found that Codex also fails in similar modes. Jigsaw trained on GPT-3 failures is able to fix similar failures generated by Codex as well in this setting. *b*) Argument transformations can be instantiated for different libraries in a manner similar to AutoPandas [9]. *c*) AST transformations and learning from user feedback extend to other languages readily (in [26], such an approach is used to learn non-trivial code refactoring in C#, SQL, Markdown, and spreadsheets). For each new API, specific transformation rules can be learnt from usage data generated by users of that API.

	Beginner	Intermediate	Advanced
Python	1	21	3
Pandas	17	8	0

Table 1: Proficiency of participants from PandasEval2 **dataset**

4 DATASETS

We perform our experiments on two different datasets [2].

4.1 PandasEval dataset PandasEval1

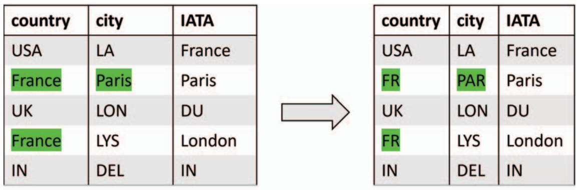
This dataset consists of 68 Python Pandas tasks. Each task can be solved using a single line of code by composing at most 2-3 Pandas functions; sometimes followed by assigning variables. This dataset was created by authors of this paper by going through queries in online forums like StackOverflow. An example task from this dataset is "For every row in df1, update 'common' column to True if value in column 'A' of df1 also lies in column 'B' of df2".

4.2 Hackathon dataset PandasEval2

This dataset consists of 21 Pandas tasks; each task can be solved by composing at most 2-3 Pandas functions, possibly followed by assigning variables, as in the PandasEval1 dataset. We posed these tasks as illustrations, in a hackathon we conducted with 25 users over 2 different sessions. The participants of this hackathon were Microsoft research fellows and interns. Table 1 presents self-reported proficiency of the users in Python and Pandas. An example illustration that shows the intent of a task is given in Figure 5. Users were asked to read such pictorial illustrations and come up with their own natural language (English) query constructions for each task to solve the problem. We then collected the queries written by the users, clustered, and annotated them to produce the PandasEval2 dataset comprising of a total 725 unique queries constructions. The task corresponding to the illustration in Figure 5, from the dataset PandasEval2, is shown below. Here `dfin` and `dfout` refer to the dataframes in Figure 5.

We note that while users provided precise and clear natural language queries in many cases, they also came up with imprecise and incorrect formulations in some cases. For instance, in the spec shown above, the query provided by `user1` is correct, whereas the one provided by `user2` is incorrect because the word "*France*" is present in the "*IATA*" column as well; Figure 5 conveys that only the "*country*" column needs to change, and not the "*IATA*" column.

[2] The datasets can be found at https://github.com/microsoft/JigsawDataset



country	city	IATA
USA	LA	France
France	Paris	Paris
UK	LON	DU
France	LYS	London
IN	DEL	IN

country	city	IATA
USA	LA	France
FR	PAR	Paris
UK	LON	DU
FR	LYS	London
IN	DEL	IN

Figure 5: Example task, part of the dataset PandasEval2**, as presented to the user during the Hackathon session.**

Since such queries were created by users interacting with the system, and users tend to make mistakes, it is useful to have such variations in the dataset. While curating the dataset, we removed natural language queries that were clearly incorrect, and retained queries that were imprecise and partially correct.

```
"task_8": {
    "queries": [
    ["replace 'France' with 'FR' in 'country'
     column and 'Paris' with 'PAR' in 'city'
     column", "user1"],
    ["In dataframe dfin, replace cells having '
     France' to 'FR' and cells having 'Paris' to '
     PR'", "user2"]
    ],
    "IO": [{
      "inputs": "dfin",
      "output": "dfout"
    }]
  }
```

Listing 1: Example json for a task in PandasEval2

As mentioned earlier, we conducted the hackathon over two sessions. We use PandasEval2_S1 to denote the dataset generated from user queries from the first session, and PandasEval2_S2 to denote the dataset generated from user queries from the second session. For each of the 21 tasks, we created semantic variations (e.g. changing constants, API arguments) of the same task. Consequently, users in the second session (PandasEval2_S2) saw different variants of the tasks when compared to users in the first session (PandasEval2_S1). Specifically, 3 tasks were exactly the same, 9 had differences in constants and 9 had changes in arguments. We introduced these variants in order to study if Jigsaw can learn from usage in the first session to improve user experience in the second session (see Section 5.2). We use PandasEval2 to denote the union of PandasEval2_S1 and PandasEval2_S2.

5 EXPERIMENTS

We evaluate Jigsaw on the two datasets introduced in Section 4, with emphasis on the following questions: *a*) How accurate is the Jigsaw system compared to the black-box PTLMs and other code synthesis methods? *b*) What is the utility and applicability of the individual Jigsaw components (in the pre-processing and post-processing modules)? *c*) Can these components benefit from feedback over time as more users interact with the system?

For the first and second questions, we evaluate Jigsaw in an offline setting, i.e., without learning from any feedback (in Sections 5.1); and present comparisons against the state-of-the-art AutoPandas framework, which generates Pandas snippets using only I/O example (in Section 5.3). For the third question, we perform a temporal study on the PandasEval2 dataset (in Section 5.2), where we leverage user feedback from the first hackathon session to update the system and measure the performance improvement in the second session. We also perform ablation studies (in Section 5.4) pertaining to our context selection sub-module. We end with a preliminary evaluation of Jigsaw on tasks pertaining to the TensorFlow API (in Section 5.5).

We consider **accuracy** as our primary evaluation metric, i.e., fraction of specifications in the dataset for which a *correct* program was synthesized. We define a program as correct if it satisfies the given I/O examples, and additionally passes a manual inspection of whether the synthesized code meets the intent of the natural language description. The manual inspection helps us reject programs that satisfy the I/O examples by overfitting on them and violate the general intent of the natural language descriptions. Note that there is inherent randomness in the output of the PTLMs, so we run every evaluation three times and report the mean accuracy (%) and standard deviation (over the runs). In some cases, we also present **task completion** metric which is the percentage of tasks correctly solved by a user (regardless of the number of queries used to solve a task) interacting with the system. Furthermore, in every case, we present the best accuracy obtained by varying the temperature parameter of PTLM $\in \{0, 0.2, 0.4, 0.6\}$.

5.1 Offline evaluation

In Table 2, we present the performance of Jigsaw on PandasEval1 and PandasEval2 datasets, with GPT-3 and Codex PTLMs. The second column of the table indicates the context selection strategy for the PTLM. For this study, we consider NO-CONTEXT (no tailored context provided for the user query; we use a default context: "`import pandas as pd`"), and TRANSFORMER (Transformer similarity based context selection, discussed in Section 3) with number of context prompts fixed as 4. Each cell in the table gives the accuracy metric with mean and standard deviation as defined above. For Jigsaw, the column titled Variable Name indicates the performance of the system using only this part of the post-processing module; and the column titled Semantic Repair indicates the performance of the system in its entirety, i.e., running Variable Name transformations followed by Semantic Repair (Argument transformations and AST-to-AST transformations).

Comparing PTLM and Semantic Repair columns, it is evident that Jigsaw improves upon the black-box PTLMs, in terms of accuracy, by 15%-40% irrespective of the context selection strategy, on both the datasets as well as on both the PTLMs. These results underscore the utility of program analysis-based augmentation of large language models.

Next, from Table 2, we find that providing useful context for the language model along with the query significantly improves upon not providing any context (comparing rows 1 vs 2, and rows 3 vs 4), across the datasets and PTLMs. It is clear that PTLM with TRANSFORMER context is better than NO-CONTEXT by a margin

		PandasEval1			PandasEval2		
		PTLM	Variable Name	Semantic Repair	PTLM	Variable Name	Semantic Repair
GPT-3	NO-CONTEXT	30.9 ± 1.2	38.2 ± 2.4	44.6 ± 3.9	8.9 ± 0.6	24.8 ± 0.9	33.6 ± 0.5
	TRANSFORMER	33.8 ± 2.4	41.7 ± 2.5	47.1 ± 2.1	6.6 ± 0.2	24.3 ± 0.8	35.1 ± 0.7
Codex	NO-CONTEXT	45.6 ± 1.2	54.9 ± 0.7	59.8 ± 3.5	26.8 ± 1.2	51.0 ± 0.6	56.8 ± 0.3
	TRANSFORMER	52.0 ± 0.7	63.7 ± 0.7	66.7 ± 0.7	31.2 ± 0.2	67.5 ± 0.5	72.2 ± 0.5

Table 2: Performance (mean accuracy ± std. deviation) after different stages of the Jigsaw **pipeline on** PandasEval1 **and** PandasEval2 **datasets.** Jigsaw **post-processing steps significantly improves upon** PTLMs **irrespective of context selection strategy. Pre-processing clearly benefits, comparing rows 1 vs 2, and 3 vs 4.**

∼ 5% (without post-processing) and up to 15% with post-processing for Codex on the two datasets. For GPT-3, TRANSFORMER context is significantly better than NO-CONTEXT on the PandasEval2 dataset and on PandasEval1, the numbers are statistically insignificant. PTLMs require some initial context in the form of examples to characterize the task to be solved, and these results underscore the importance of having a pre-processing module in Jigsaw.

Finally, from Table 2, we also observe the effectiveness of the individual post-processing modules of Jigsaw, as discussed below. Note that, for these results, we seed our AST-to-AST transformations using a small dataset collected from StackOverflow questions. Later, in Section 5.2, we show that these numbers can be significantly improved by learning transformations from usage over time.

Variable Name transformations: PTLMs make variable referencing errors (as noted in Section 2) because of its implicit bias towards common dataframe names such as `df, df1, df2, dfout` and also because users tend to not specify variables explicitly in their queries. We find that this simple post-processing module gives an improvement of 10%-30% for Codex and 10%-15% for GPT-3.

Semantic Repair: We see that the semantic repair post-processing module improves absolute performance of Codex by ∼ 5% and of GPT-3 by 6%-11%. This underscores the significance of using program analysis techniques to augment language models that do not have inherent understanding of code semantics. Recall (from Section 3) that Semantic Repair consists of Argument transformations and AST-to-AST transformations sub-modules. We find that, using just the Argument transformations (without AST-to-AST transformations), improves absolute performance of the system by 5%-9% and 3%-5% for GPT-3 and Codex respectively (not shown in Table 2). Similarly, using AST-to-AST transformations alone (without Argument transformations), we obtain improvement of up to 3.5% for GPT-3 and 1.3% for Codex (not shown in Table 2).

We find that our post-processing steps are reasonably fast; time taken by Jigsaw is primarily bottle-necked by the inference times of PTLM APIs. Specifically, on average getting output from Codex takes ∼ 7 seconds while our post-processing module takes < 3 seconds. Similarly, on average, GPT-3 takes 30-40 seconds for different context sizes while post-processing finishes in < 10 seconds.

5.2 Temporal evaluation

In this section, we evaluate Jigsaw on its ability to learn and improve with user feedback. We perform this evaluation on the PandasEval2 dataset. Recall that the hackathon was organized over two separate sessions; so, we use the submissions and feedback for tasks in the first session, corresponding to the PandasEval2_S1 dataset, to (a) update our context bank, (b) learn AST-to-AST transformations, and (c) evaluate Jigsaw on the PandasEval2_S2 dataset consisting of variants of the tasks in PandasEval2_S1 (as described in Section 4.2).

Updating pre-processing module: We follow Algorithm 1 (with $\epsilon_{\text{CODE}} = 25$, $\epsilon_{\text{BANK}} = 0.15$) and filter out badly written queries from the first session users. Note that, for the 3 tasks that are identical in the two sessions, we do not make any updates to the context bank. We denote our seeded context bank that was used in the first session (containing 243 question-answer pairs) with CS1 and the updated context bank updated resulting from Algorithm 1 with CS2 containing 371 (243 seeded + 128 new) question-answer pairs.

Updating post-processing module: We follow the procedure described in Section 3.4 to learn AST-to-AST transformations from session one data along with seeded data. We use TS1 to denote transformations seeded during session one and TS2 to denote the new/updated transformations learned from session one data.

We compare the performance of Jigsaw on PandasEval2_S1 (with CS1 and TS1) against PandasEval2_S2 (with baseline CS1 and TS1, as well as with the updated context bank CS2 and transformations TS2) in Table 3. Each cell in the table is the mean accuracy and standard deviation on the corresponding dataset. Two observations are in order.

(1) Learning helps improve Jigsaw**.** It is evident that the performance of Jigsaw on the PandasEval2_S2 dataset with the default CS1-TS1 setting (column 3) is significantly lower than that of the updated CS2-TS2 setting (column 4) for both the PTLMs. Accuracy of the system with GPT-3 improves by over 30% due to the updated modules; even with Codex, which already performed quite well on all datasets, we still improve by ∼ 15% with updates.

(2) Second session was in general more challenging. We also observe that the performance on PandasEval2_S2 with the default CS1-TS1 setting (column 3) is significantly lower than that on PandasEval2_S1 with the same setting (column 2). This is because in general the second session was more challenging; partly due to the higher percentage of queries on difficult tasks, and the semantic differences in tasks across the two sessions. But when we use the updated the context and transformations banks, we find a drastic improvement in the performance on PandasEval2_S2, as highlighted in (1) above. This illustrates that Jigsaw has the ability to improve from user feedback, regardless of the PTLM used.

Finally, we also look at the task completion metric (described in the beginning of Section 5), to assess how the performance gains of learning from feedback translated to user experience during the hackathon. In session one, users we able to solve only 71% of the tasks on average; however, in session two, users were able to solve 82% of the tasks on average, thus making the experience of the Jigsaw system more productive with the updates.

	PandasEval2_S1	PandasEval2_S2	
	CS1-TS1	CS1-TS1	CS2-TS2
GPT-3	45.9 ± 0.4	35.1 ± 0.8	67.2 ± 0.3
Codex	75.1 ± 0.5	69.0 ± 0.7	84.4 ± 0.8

Table 3: Performance (mean accuracy ± std. deviation) of Jigsaw **without** (CS1-TS1) **and with** (CS2-TS2) **learning context bank and transformations from user feedback on the** PandasEval2 **dataset. Learning helps improve accuracy significantly, comparing columns 3 and 4.**

5.2.1 Analyzing learned AST-to-AST transformations. We present some of the learned AST-to-AST transformations applied to code snippets produced by GPT-3 in Table 4. The transformations were learned using the clustering and perturbing technique outlined in Section 3.3. We see that the code fixes are interpretable and they solve common semantic problems in the outputs of PTLMs. Please refer to supplementary material for details of the precise AST-to-AST transformations learnt corresponding to first two rows of Table 4.

For instance, consider the rule implied in the first row of Table 4, which is of inserting ~ (bitwise not operator) inside subscript. This transformation, learned using the cluster of *diverse* code snippets in Listing 2, is fairly general (this is one of the clusters obtained by running our clustering technique on seeded and session one data). On the other hand, consider the last row of Table 4, which was learned using the cluster of code snippets in Listing 3. Since the clustered snippets follow a similar structure, the learned transformation works only when a new snippet has exactly the same logical conditional operators in the specific order. Thus, the quality of the learned transformations depends on the quality of the clustering and of the code snippets themselves, and we expect that more usage data positively influences the overall quality.

```
# Task-1
dfout = df.loc[df.isnull().any(axis=1), :] #incorrect
dfout = df.loc[~df.isnull().any(axis=1)] #correct
# Task-2
df_p = df_p.loc[df_per["Name"].str.contains("Ch")] #incorrect
df_p = df_p.loc[~df_per["Name"].str.contains("Ch")] #correct
```

Listing 2: Cluster of code snippets from two different tasks that yields the Bitwise-Not transformation in Table 4.

5.3 Comparison to AutoPandas

AutoPandas (AP) [9] is a Pandas program synthesis engine capable of generating programs with two or three Pandas functions. It uses *generators* for enumerating over the Pandas API and guides the search with the help of Graph Neural Networks (GNNs) which operate on the input-output (I/O) dataframe(s) and returns the most likely function sequences and arguments.

In contrast, we make use of multi-modal specification (both natural language query and I/O examples). Programming by examples is known to have ambiguous under-specifications [19, 32]. From our experience this issue is exacerbated for large APIs that provide multiple ways for achieving similar functionalities. For instance, consider the specification in Figure 1. If we only consider the I/O example for the given task, we can find many trivial solutions that just drop or select certain rows of dataframe.

We evaluate AP on our PandasEval1 and PandasEval2 datasets. As discussed in Section 3, AP does not support series operations, column assignments and dictionary or list generators, many of which are necessary in Pandas workflows. So, out of 68 tasks in the PandasEval1 dataset and 21 tasks in the PandasEval2 dataset, only 20 and 7 are covered by the AutoPandas framework respectively. Hence, we compare Jigsaw (instantiated with the Codex PTLM) against AP only on these 27 tasks and use a timeout of 3 minutes. In the first row of Table 5, we see that Jigsaw clearly outperforms AutoPandas even in the restricted subset solvable by AP. This is because 16 of the 27 tasks are under-specified if only I/O examples are used and AP returns over-fitting solution on many of these tasks; this highlights the necessity of multi-modality.

We also run Jigsaw on the AP dataset [9], where all tasks are supported by AutoPandas and I/O examples are sufficient. This dataset has been sourced from StackOverflow posts. Since Jigsaw uses text as the primary input, we add natural language descriptions in these posts for querying Codex. The results are in the second row of Table 5; while Codex alone is inferior to AP, Jigsaw (with Codex) performs better than AP.

5.4 Ablation study

In both the offline and temporal evaluations presented in the previous subsections, we fixed the number of context prompts to 4 and TRANSFORMER as the context selector in the pre-processing module. In this ablation study, we ask if the performance of Jigsaw is sensitive to these choices, and provide justification for the same. All experiments in this section are carried out with the same setting as that of Section 5.1.

Table 6 compares the performance of Jigsaw with two different context selection strategies, namely, TFIDF and TRANSFORMER. We find that the transformer context selector is slightly better, but more importantly, that the performance of Jigsaw is not sensitive to the selection strategy. Table 7 compares the performance of Jigsaw with different number of context prompt examples, i.e., 1, 4, and 8. Our experiments show that while there isn't a significant difference between the performances of 4 prompts vs. 8 prompts, both perform better than using just 1 prompt. Again, Jigsaw is relatively robust to these choices.

Finally, note that all variations of these choices, for the number of prompts as well as the selection strategy, outperform the NO-CONTEXT setting (see Table 2); this further underscores the utility of the pre-processing module.

5.5 Beyond Pandas

To test the generality of Jigsaw, we did a preliminary evaluation with 25 TensorFlow [6] tasks sourced from TF-coder [40] and online forums like StackOverflow. We setup the pre-processing module of Jigsaw similar to the offline evaluation, by creating a context bank of 25 prompts from documentation pages. We reuse the Variable Name module and do a *what-if* analysis for argument and tree transformations manually. Table 8 shows the performance of Jigsaw on the TensorFlow dataset. As seen from the table, Codex alone is able to solve only 8 of the 25 tasks, variable transformation

[3] Interestingly, this missing parenthesis mistake is quite common and frequented even by humans! See blog post [3] and StackOverflow question [4].

Code Before	Code After	Semantic Explanation
`out=data[data.index.isin(test.index)]`	`out=data[~data.index.isin(test.index)]`	Adding bitwise not inside subscript
`df=df[df['foo']>70|df['foo']<34]`	`df=df[(df['foo']>70)|(df['foo']<34)]`	Parenthesizing mistake[3]
`out=df.iloc[0,"HP"]`	`out=df.loc[0,"HP"]`	Changing `iloc` to `loc`
`dfout=df1.append(df2,ignore_index=True)`	`dfout=df1.append(df2)`	Dropping the last keyword argument
`dfout=dfin.duplicated()`	`dfout=dfin.duplicated().sum()`	Computing sum of series using `.sum()`
`train=data.drop(test)`	`train=data.drop(test.index)`	Adding `.index` in first argument (of drop)
`dfin=dfin["A"].rolling(window=3).mean()`	`dfin["A"]=dfin["A"].rolling(3).mean()`	Reassign back to the column
`dfout=dfin[(x<40)|(y>53)&(z==4)]`	`dfout=dfin[((x<40)|(y>53))&(z==4)]`	Giving precedence to bitwise-or

Table 4: Applications (Code After) of learned transformations on code snippets produced by PTLM (Code Before).

```
#Task-1
dfout = dfin[(dfin["gamma"]<40)|(dfin["gamma"]>53)&(dfin["alpha"]==4)] # incorrect
dfout = dfin[((dfin["gamma"]<40)|(dfin["gamma"]>53))&(dfin["alpha"]==4)] # correct
#Task-2
dfout_per = dfin_per.loc[(dfin_per["alpha"]<140)|(dfin_per["alpha"]>159)&(dfin_per["beta"]==103)] # incorrect
dfout_per = dfin_per.loc[((dfin_per["alpha"]<140)|(dfin_per["alpha"]>159))&(dfin_per["beta"]==103)] # correct
```

Listing 3: Cluster of code snippets from two different tasks that yields the precedence transformation in Table 4.

	AutoPandas [9]	PTLM	Jigsaw
Subset of Jigsaw datasets	16/27	20/27	23/27
AutoPandas dataset	17/26	15/26	19/26

Table 5: Number of tasks solved by Jigsaw and AP on a subset of our dataset supported by AP and their dataset.

	Context	PandasEval1	PandasEval2
GPT-3	TFIDF	46.5 ± 4.8	32.4 ± 0.5
	TRANSFORMER	47.1 ± 2.1	35.1 ± 0.7
Codex	TFIDF	69.1 ± 2.4	70.1 ± 0.1
	TRANSFORMER	66.7 ± 0.7	72.2 ± 0.5

Table 6: Ablation study: Performance of Jigsaw with two context selection strategies.

	# Prompts	PandasEval1	PandasEval2
GPT-3	1	47.5 ± 1.8	34.9 ± 0.9
	4	47.1 ± 2.1	35.1 ± 0.7
	8	48.0 ± 2.5	32.9 ± 0.6
Codex	1	62.3 ± 0.7	71.8 ± 0.5
	4	66.7 ± 0.7	72.2 ± 0.5
	8	66.2 ± 1.2	72.4 ± 0.9

Table 7: Ablation study: Performance of Jigsaw with different number of context prompts.

PTLM	Variable Name	Semantic Repair
8/25	15/25	19/25

Table 8: Preliminary results of Jigsaw with TensorFlow API.

improves the performance to 15 tasks. We manually compare the code outputs to the expected output, to check if argument and tree transformations can be learnt. Based on this analysis, we find that Semantic Repair can potentially improve the performance to 19 tasks. We show some examples below. For the query “Given a tensor in1, replace all instances of 1 with 0”, PTLM outputs the following:

```
tf.where(x == 1, 0, x)
```

The correct code for this query, synthesized by Jigsaw using variable transformation, is shown below:

```
tf.where(in1 == 1, 0, in1)
```

For the query “Given a tensor in1 and a tensor of indices ind, get the sum of elements present at indices in ind from tensor in1. ”, the PTLM outputs the following incorrect code:

```
tf.gather(in1, ind)
```

The correct code, shown below, can be synthesized by Jigsaw with a learnt AST-to-AST transformation, if sufficient data points are collected from usage:

```
tf.reduce_sum(tf.gather(in1, ind))
```

In summary, this shows that the proposed pre-processing and post-processing modules are useful, and can be generalized to other libraries and programming languages as well.

6 THREATS TO VALIDITY

Our data sets have been created by manually inspecting internet forums like StackOverflow. We tried to cover the common programming patterns in Pandas. However, they are not representative of all Pandas programs in the wild.

We designed the PandasEval2_S1 and PandasEval2_S2 datasets by collecting data from two sessions of a hackathon, as a proxy for the real-world setting, where large software teams are working on the same project with similar tasks, allowing Jigsaw to learn and improve over time. We varied the tasks between the two sessions, so as to simulate variants of tasks. However, the variations we introduced may not representative of variations of tasks in the real world. Our study had only 25 participants; evaluating whether the productivity of developers is enhanced in a statistically significant manner in a large scale deployment of Jigsaw is beyond the scope of this paper.

When comparing Jigsaw to AutoPandas, Jigsaw takes as input both the natural language description and the I/O examples, while AutoPandas only takes the I/O examples as inputs. Hence, Jigsaw has more information about the tasks than AutoPandas. Jigsaw takes less than a minute per task and we use a timeout of three minutes for AutoPandas. Although higher timeouts might improve the performance of AutoPandas (10-15 minutes [9]), they are not compatible with the interactive user experience that we are aiming for. Whether AutoPandas solved a task correctly or not is determined by manual inspection and is susceptible to human errors.

7 RELATED WORK

The literature on using machine learning for program synthesis is vast [8, 17, 18, 21, 22, 25, 28, 35, 41] and we restrict to works which are closest to Jigsaw (synthesizing code for large APIs using large models and multi-modal specifications). These works can be classified into the following categories: 1) designed for large APIs but do not use large models, 2) based purely on large models with no multimodal specification, and 3) multimodal synthesis for small APIs. Details follow:

(1) The TDE [20] system for Java relies on rich type information (which is absent in Pandas) and fails to generate argument combinations that are absent from its corpus. AutoPandas [9] generates Pandas code exclusively from input-output (I/O) examples using a combination of GNNs, which predict function sequences, and enumerative search. TF-coder [40] uses both natural language descriptions and I/O examples to generate TensorFlow code. Both of them use small specific models (as opposed to large generic models like GPT-3) and lack mechanisms to incorporate user feedback.
(2) GPT-3 [10] while trained on web has shown inspiring capability on synthesizing code. Models have also been explicitly trained on code with documentation [7, 11, 15]. In particular, Codex [11], that is part of GitHub Copilot, generates Python code from natural language descriptions. Syncromesh [31] uses build context prompts for PTLMs similar to how Jigsaw does in pre-processing. Additionally, they propose Constrained Semantic Decoding for generating code while respecting syntactic and semantic constraints. However, the expressiveness of the constraints is restricted and more work is needed to model constraints that occur in practice. Spider [5, 45] is a text-to-SQL competition where many tools compete [38, 42].
(3) Rahmani et al. [34] use the outputs of GPT-3 to guide a component based search. Their approach is evaluated only on small DSLs such as regular expressions and CSS selectors and they do not learn from user feedback. Manshadi et al. [24] and Raza et al. [36] synthesize string transformations. WebQA [12] synthesizes programs to extract information from webpages. Regel [13] and Ye et al. [44] synthesizes regular expressions. Mars [14] synthesizes data wrangling operations. These techniques have not been demonstrated at the scale of Pandas that has hundreds of operations.

Jigsaw fixes the output of PTLM and is hence related to work on program repair like Refazer that learns code transformations from edits used to fix programs [16]. Jigsaw's interface is inspired from B2's [43] interface that augments visualizations to notebooks.

8 CONCLUSION AND FUTURE WORK

Jigsaw is the first tool for synthesizing code for large APIs like Pandas that leverages the advancements in PTLMs. The key contribution of Jigsaw lies in the post-processing steps that drastically improve the quality of the code generated by PTLMs like GPT-3. In particular, the multimodal synthesis of Jigsaw outperforms both the baselines that exclusively use PTLMs and those that exclusively use I/O examples for program synthesis. However, several challenges remain before we can have a true "pair programmer" experience with PTLMs and we discuss a couple of them.

First, in this paper, the quality of the synthesized code is largely determined by the I/O examples. However, in practice, code quality is more nuanced than correctness on unit tests. Ideally, the synthesized code should have high performance, should not have security vulnerabilities [29], and respect licensing attribution [4].

Second, Jigsaw focuses on multi-modal specifications with natural language intent and I/O examples. However, even multi-modal specifications can be weak or ambiguous, and would need to be refined using richer specifications like preconditions, postconditions, invariants, bounds on resource usage like time and memory, etc., to obtain the intended code.

Acknowledgement. We thank Dhvanil Sanghvi for helping us perform a preliminary evaluation of Jigsaw with Tensorflow (reported in Section 5.5), and Arjun Radhakrishna for helping us with PROSE and Refazer related queries.

[4] https://www.wired.com/story/github-commercial-ai-tool-built-open-source-code/

REFERENCES

[1] [n. d.]. GitHub Copilot · Your AI pair programmer. https://copilot.github.com/

[2] [n. d.]. Jupyter. https://jupyter.org/

[3] [n. d.]. Parenthesis Blog. https://www.roelpeters.be/cannot-compare-a-dtyped-object-array-with-a-scalar-of-type-bool/

[4] [n. d.]. Parenthesis StackOverflow. https://stackoverflow.com/questions/38252423/python-error-typeerror-cannot-compare-a-dtyped-float64-array-with-a-scalar-o

[5] [n. d.]. Spider 1.0: Yale Semantic Parsing and Text-to-SQL Challenge.

[6] [n. d.]. TensorFlow. https://www.tensorflow.org/

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *ArXiv* abs/2108.07732 (2021).

[8] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[9] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 168:1–168:27.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).

[12] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 328–343.

[13] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multimodal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 487–502.

[14] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal multi-layer specification synthesis. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 602–612.

[15] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online.

[16] Reudismam Rolim de Sousa, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjoern Hartmann. 2016. Learning Syntactic Program Transformations from Examples. *CoRR* abs/1608.09000 (2016).

[17] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 990–998.

[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 420–435.

[19] Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering* 45, 137 (2016), 3–15.

[20] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (2018), 1165–1177.

[21] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[22] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 436–449.

[23] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

[24] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. 2013. Integrating Programming by Example and Natural Language Programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, Marie desJardins and Michael L. Littman (Eds.). AAAI Press.

[25] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 187–195.

[26] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. ACM. https://www.microsoft.com/en-us/research/publication/on-the-fly-synthesis-of-edit-suggestions/

[27] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[28] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[29] H. Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and R. Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *ArXiv* abs/2108.09293 (2021).

[30] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True Few-Shot Learning with Language Models. *ArXiv* abs/2105.11447 (2021).

[31] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*.

[32] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

[33] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[34] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Dan Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal Program Inference: a Marriage of Pre-trained Language Models and Component-based Synthesis. In *OOPSLA*.

[35] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 419–428.

[36] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, Qiang Yang and Michael J. Wooldridge (Eds.). AAAI Press, 792–800.

[37] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084

[38] Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, Kristina Toutanova, Anna

Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, 311–324.

[39] Gerard Salton and Michael J McGill. 1986. Introduction to modern information retrieval. (1986).

[40] Kensen Shi, David Bieber, and Rishabh Singh. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. *CoRR* abs/2003.09040 (2020).

[41] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 398–414.

[42] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7567–7578.

[43] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 152–165.

[44] Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig, and Greg Durrett. 2019. Sketch-Driven Regular Expression Generation from Natural Language and Examples. *CoRR* abs/1908.05848 (2019).

[45] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 3911–3921.

[46] Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *International Conference on Machine Learning*.