Parametric API Summarization : Template Distillation from LLMs

Naman Jain, Aditya Parameswaran, Koushik Sen, Ion Stoica

Data Science APIs are complex!

- Data science use cases growing rapidly
- Yet, APIs like Pandas, SkLearn are challenging for novice users
- For example, Pandas provides over 700 API methods

Data Science APIs are complex!

- Each API method provide multiple arguments, each allowing multiple argument types

- The space of possible argument combinations is therefore exponential

`DataFrame.replace(to_replace=None, value=_NoDefault.no_default, *, inplace=False, limit=None, regex=False, method=_NoDefault.no_default)` [source]

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically.

This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters: **to_replace** : ***str, regex, list, dict, Series, int, float, or None***

How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

Program Summarization For Explainability

Summarize the following pandas program imperatively in a line `df.replace({'city' : 'NYC', 'population': ['-', '?']}, {'city' : 'NY', 'population': np.nan})`

The program replaces the values 'NYC' in the 'city' column and '-' and '?' in the 'population' column of a Pandas dataframe with 'NY' and NaN respectively.

Language models can generate useful and fluent summaries from programs!

Issues with language model-based summaries

Language models are generative models of sequences that at best only have shallow understanding of program semantics.

1) They can make mistakes

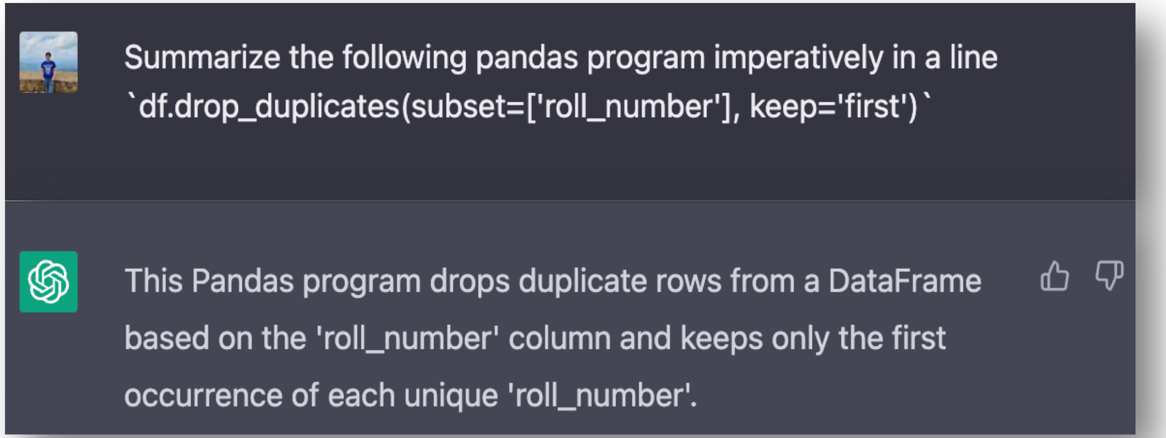
2) They are challenging to control

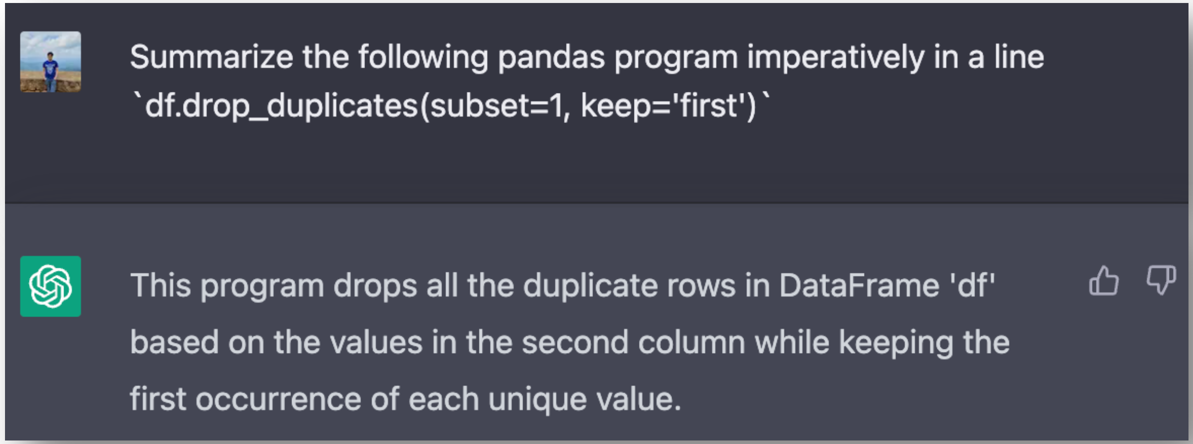
Issue 1.a - Robustness

Language models are generative models of sequences that at best only have shallow understanding of program semantics.

1) They make mistakes

- not robust to small perturbations in programs

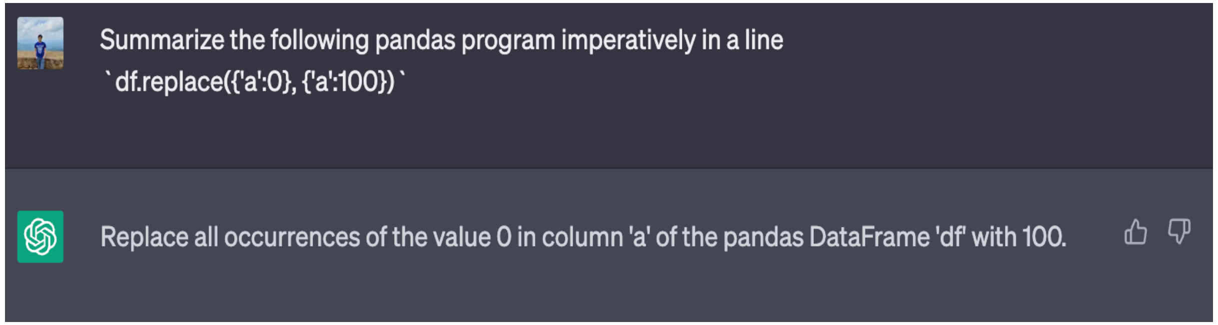


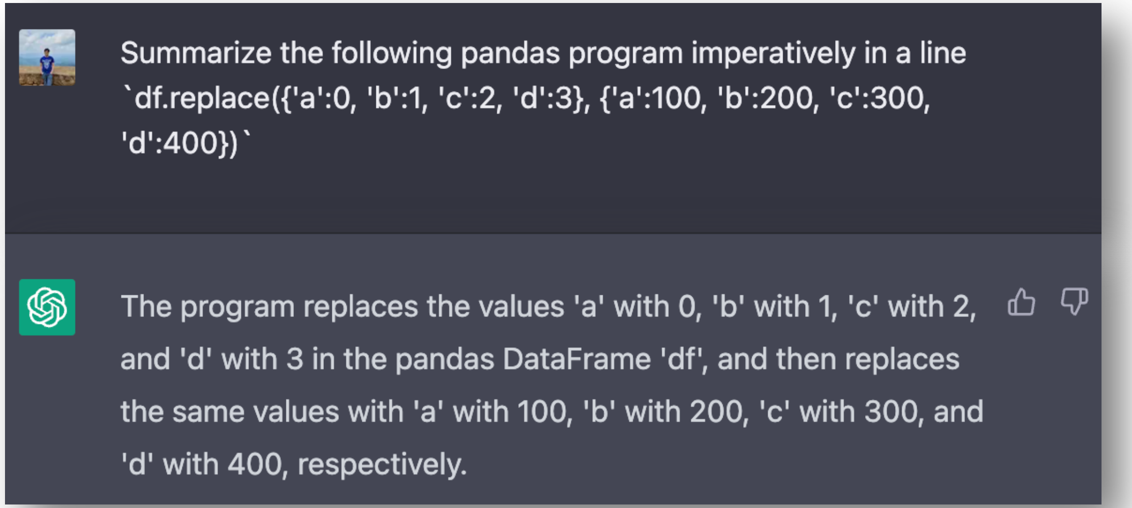


Issue 1.b – Compositional Generalization

Language models are generative models of sequences that at best only have shallow understanding of program semantics.

1) They make mistakes
   - do not generalize compositionally





Issue 2 - Controllability

Language models are generative models of sequences that at best only have shallow understanding of program semantics.

2) They are challenging to control

a) These systems generate summary in a one-shot fashion
b) give very little end-user control on how the output should look

Our Approach - Fusing API knowledge with LLMs

- LLMs are stochastic and noisy!
- `use symbolic knowledge to improve capabilities and provide guardrails

- Previous work (Jigsaw)
  - combined synthesis and repair techniques to improve Pandas code generation

- API programs contains methods with well defined structure and semantics

API Summarization – An Alternative View

- API programs comprise of specific methods with
  - well-defined structure
    - argument combinations
    - return types

Given this structure and semantics in API programs, can we discover an intermediate templatized natural language describing the programs

Outline

Define Parametric Templates

Learning templates by combining LMs with API knowledge

Data Driven Template Verification

Parametric Templates - Example

```
df[df['score'].isin(range(5,10))]
```

Select the rows where value in column score lie in the integers between 5 and 10 (exclusive)

Parametric Templates - Example

```
df[df['score'].isin(range(5,10))]
```

Select the rows where value in column score lie in the integers between 5 and 10 (exclusive)

- Template[subscript, [caller - df, arg - expr]]
  - Select the rows VAR1
  - VAR1 = Summary(`df['score'].isin(range(5,10))`)
- Template[isin, [caller - `df['score']`, value - range(5,10)]]
  - where values in column score lie in VAR1, where
  - VAR1 = Summary(`range(5,10)`)
- Template[range, [start - int, end - int]]
  - the integers between start and end(exclusive)

Parametric Templates

For any API function with signature $t_1, t_2 \ldots, t_n$

Parametric template T is a sequence $\{x_1, x_2, \ldots, x_m\}$, where

- $x_i = w_i$ (word) or
- $x_i = F_i$ (a function from arguments to words)

Parametric Templates - Example

Consider the program

```
df.replace({'a':1, 'b':2}, {'a':3, 'b':4})
```

Replace the values 1 in a and 2 in b with 3 and 4 respectively

The template is

Replace the values F^1 with F^2 respectively

F^1 = `"and ".join([value + " in " + key for key, value in arg1.items()])`

F^2 = `"and ".join([arg2.values()])`

How to come up with such templates

Writing these templates manually is hard

requires domain expertise	templates are fuzzy and hard to manually annotate	numerous functions and arguments!

Here we observe that it is hard to write them but easy to verify and modify

Learning templates

- We automatically learn parametric templates from a corpus of API snippets and their summaries (written manually or from language models)

- We learn these templates by
  - using dynamic programming on constituency parse trees of summaries
  - bottom-up program-synthesis of hole functions
  - word and phrase similarities

Learned Templates

```
df.replace({'country': {'Germany':'GER', 'France':'FRA'}})
```

Replace the values F^1 in column F^2 with F^3 respectively

$F^1(\mathrm{P})$ = `'Germany' and 'France'`

$F^2(\mathrm{P})$ = `'country'`

$F^3(\mathrm{P})$ = `'GER' and 'FRA'`

Learned Templates

```
df.replace({'a':1, 'b':2, 'c':3}, {'a':100, 'b':200, 'c':300}
```

Replace the values F^1 with F^3 respectively

$F^1(\mathrm{P})$ = `1` in `'a'`, `2` in `'b'`, `3` in `'c'`

$F^2(\mathrm{P})$ = `100, 200, 300`

Learned Templates

```
df.dropna(subset=['score1', 'score2', 'score3'], thresh=2)
```

Drop the rows in df having atleast F^1 nans in the F^2 columns

$F^1(\mathrm{P})$ = `2`

$F^2(\mathrm{P})$ = `'score1'`, `'score2'`, and `'score3'`

Data Driven Verification

- How to evaluate quality of templates?

- Utilizing LMs to evaluate the quality of summaries generated from the templates
  - measure perplexity of generated summaries
  - recovering the API method back from summaries (bi-directional consistency!)