Assignment 3

Group 19

Project

Name: ArviZ
URL:https://github.com/arviz-devs/arviz

What it is:

ArviZ is a Python package that enables data exploration and analysis of Bayesian models. It includes functions for posterior analysis, data storage, model checking, comparison and diagnostics.

Onboarding

Did it build and run as documented?
Our first choice of project was Pandas.While building a Pandas clone, we ran into several issues, including missing C extensions and incomplete dependency installation. The required dependencies were scattered across the documentation, making it difficult to track them all down. For instance, using python 3.10 was only mentioned in the MacOS section. We also encountered build tool configuration issues related to Visual Studio and environment variables, such as with some NVIDIA settings. Some group members could not resolve these issues. Therefore, we decided to change project.

The onboarding for ArviZ on the other hand was more or less as smooth as it can be. The installation was little more than just cloning the repo, creating a virtual environment and running the install script + pip install on the requirements doc. There were some confusion about which requirements and dependencies to install in order to run all the tests correctly since they had a separate file for those, but that was quickly sorted out.

Complexity

1. What are your results for five complex functions?

- **convert_to_inference_data()**

  Given the differences in calculations, the manual count and lizard gave the same result. Lizard gives 33 while the manual one gives 20, however, there are 14 exit points which means the total will give the same result when accounting for lizard not counting early

exits. It could thus be said that the high amount of exit points hide the actual complexity of the function when using the manual method

- **plot_trace()**

  Both lizard and the manual count result in a CCN of 68. We counted 68 decision points and 2 exit points. However, we counted an ‘assert’ as an exit point because this could terminate the function but lizard does not seem to count asserts as exit points, so lizard counted some statement as a decision point which we did not count.

- **plot_separation()**

  The tool count and the manual count give similar results. The tool count gives CCN of 15 but the manual count gives a CCN of 10

- **summary()**
  The tool complexity (using lizard) is 54. Manual counting gives 55.

- **concat()**

  The tool count and the manual count give a similar result. lizard gives 98 while the manual count gives 83.

2. Are the functions just complex, or also long?
All functions are relatively long, but there is an amount of correlation between the longer functions and a higher CCN. The two shorter functions, at ca 70-80 LOC, are also the ones with the lowest CCN. However, it’s worth noting that the longest function, while it is really complex, is not the *most* complex.

Function	LOC	CCN
Convert_to_inference_data()	82	33
plot_trace()	292	68
plot_separation()	72	15
summary()	354	54
concat()	232	98

3. What is the purpose of the functions?

- **convert_to_inference_data()**

  The purpose of the function is to take an object and, depending on what it is, send it to the correct converter. This means that it inherently will have a lot of branching, as it will have to determine what type the object is, most of the function is just a long if, elif, elif… string covering different cases. Some cases do however have more complex logic to determine subtypes, file types etc.

- **plot_trace()**

  There are several versions of plot_trace in the code, this one uses the bokeh backend for plotting. It plots how data changes over time, with options for plotting density, individual data points, divergences and reference lines.

- **plot_separation()**

  The purpose of the function is to visualize model predictions of binary outcome models.

- **summary()**
  It creates a data frame with summary statistics for the provided inputs.

- **concat()**

  The purpose of the function is to concatenate InferenceData objects of different formats into one. The high cyclomatic complexity is directly related to this purpose: each branch corresponds to handling a specific case or validation (e.g., input validation, dimensional checks, merging attributes, error handling). While these branches make the function more versatile, they also increase its overall complexity.

4. Are exceptions taken into account in the given measurements?

We counted an exception as a branch, and some rudimentary testing showed that this is the case for lizard aswell. Lizard does however not appear to take into account early returns. This means that functions with a lot of early returns will have a much lower CCN score when manually counted.

5. Is the documentation clear w.r.t. all the possible outcomes?

- **convert_to_inference_data()**

  While the documentation is clear regarding the different outcomes depending on what type the object is, it does not provide details regarding exceptions that it might raise.

- **plot_trace()**
  plot_trace with the bokeh backend has next to no documentation. However, the backend agnostic version is well documented. Each parameter's effect on the plot is described. Either the bokeh figure is returned or, if divergences are to be plotted but no divergence data is provided, the function fails.

- **plot_separation()**

  The function documentation is detailed but there is definitely some room for improvements regarding exception cases.

- **summary()**
  The function documentation is brief and it's hard to understand what exactly the function does.
- **concat()**

  The function's docstring explains its general behavior, parameters, return values, and provides examples. However, it does not describe in detail how every branch or condition (for instance, different error cases or the differences arguments) affects the outcome

Refactoring

- **convert_to_inference_data()**
  Since the main structure of the function is to go through many different cases for a singular object, and then performing similar tasks on it, the plan was to extract all logic within these outermost cases to its own function. For instance, any preprocessing and any sublogic for sub-cases was extracted. There was also a couple of cases of code duplication that could be simplified. Some cases were ignored as they were too small to extract. If the pre-processing is a single line, it doesn't really make sense to extract it to its own function. In general, the result is a function which has a more clear structure, but which is more abstract. An example of this is that it is very clear that when the object is a string, the object is treated as a file and is converted, however the logic as for how this is done, any pre-processing for this etc, is now abstracted away. It's worth mentioning that the function is still very much an if-elif string but now it's much clearer what each case does. Extracting more would reduce the readability of the function to an unnecessary level considering what the function is meant to accomplish.

  The refactoring can be viewed in the branch issues/22 or by:

  git diff 0166bde f06ce83

In the end, the CCN reported by lizard was reduced from 33 to 20.

- **plot_trace()**
  The function is long and complex, but it is easy to extract some of the logic into separate functions. For instance, getting the backend config, configuring figure size, getting plot properties, setting default kwargs values, plotting the reference lines and plotting the divergences. These parts of the function can easily be extracted into their own functions, reducing the CCN by roughly 40% without any other major changes.

  The refactored code can be seen in issues/37 or by running:

  git diff bada5cb4 9eb75056

  This refactor reduces the CCN from 68 to 36

- **plot_separation()**

  The plan for refactoring this function is to split it up into 3 functions. The idea is to create two new functions where one function will validate the inputs and one function will process the validated input and return the y, y_hat and label_y_hat values. These two functions will basically be helper functions for the plot_seperation function.

  The estimated impact of this refactoring is that the CC will be lower since the plot_seperation function will contain less branching. Another impact of refactoring will be that the code will be clearer to read and more maintainable. To see the refactoring you can run the following command

  The refactoring can be viewed by running the following command:

  git diff 0166bde1bac4ae84632cbe7df60f1de2520cd5d
  889dd550bb90b3e9c264302adb33f23474586cf5

  In the end, the CCN reported by lizard was reduced from 15 to 2.

- **summary()**
  The plan for refactoring is to extract the validation of the data into a separate function. This makes sense functionally as the validation of the data is a complexity heavy part of the function while not actually performing any calculations on the data.

  git diff e5575b5 4d8b611

  The CCN of the function after splitting dropped from 54 to 33 (reduced by 39%). The CCN for the validation function is 22.

Further optimizations can be done by splitting the function at the following lines:
“if stat_funcs is not None:”
“if extend and kind in ["all", "stats"]:”
“if circ_var_names:”
“if kind in ["all", "diagnostics"] and extend:”
“if circ_var_names and kind != "diagnostics" and stat_focus == "mean":”
This would only leave the summary_df, the return variable, and its calculations in the original function while the other child functions, which only affect the output if their respective flags are set, would do their calculations in separate functions.

- **concat()**

  The function is that it first validates the inputs. Then it concatenates the InferenceData objects either over unique groups or dimensions which depends on whether the dim argument has been passed to the function or not. Therefore, I split the concat function into 4 functions by defining 3 new helper functions. The first one validates the inputs, the second one concatenates over the groups while the third one concatenates over the dimensions. This reduces the CCN of the function, making it clearer and abstracting several details.

  git diff 597981641f980630348c639a5ec74050e5f722e3 cdda2ba32019236897139b25586db47b7020d668

  In the end, the CCN reported by lizard was reduced from 98 to 21.

Coverage

Tools

The coverage tool we chose was coverage.py. Our experience with using coverage.py was as good as it can be. The tool is very easy to integrate into the build environment since it basically only requires installation with pip. Besides that, the tool is very easy to use and straightforward and it also has a good amount of documentation. For example, the command used for testing the coverage and outputting the report into html format was very easy to find. The html page in itself was also very user-friendly and easy to navigate through.

Our own coverage tool

An example of our own implementation can be viewed here or by running the following git diff:

git diff e5575b585df36fe55076bf15fb8debfa0bbd5635
52291ea4b47d6669a4126030dc77f8c25da24969

What kinds of constructs does your tool support, and how accurate is its output?

Our tool supports if, else if, and else constructs. It does not support ternary operators. Its output is very accurate compared to coverage.py. In specifically the plot_seperation() function the output of the DIY tool is the same as the coverage.py tool. More details regarding implementation can be viewed below.

Evaluation

1. How detailed is your coverage measurement?

Our tool covers the basic if statement very well. However our tool does not support ternary operators. In order to support this one could easily add a separate if statement below the ternary one with the same conditions, but with the sole purpose of setting the branching variables.

We have also decided to take exceptions into account, and have added a branch_id for those aswell.

2. What are the limitations of your own tool?
Our tool only checks for the branching, and does not take into account the number of lines that each branch has. This is in contrast to how coverage.py counts coverage as it counts the number of lines As mentioned earlier, the tool does not account for ternary operators at the moment. Furthermore the tool is not automatised in the sense that if new branches are added the one has to manually go in and add flags.

3. Are the results of your tool consistent with existing coverage tools?

- **convert_to_inference_data()**
- **plot_trace()**
  The results of the DIY coverage is close to coverage.py. Before, 83% vs 87%, after increasing coverage: 92% vs 93%.
- **plot_separation()**
  The results of our tool are consistent with the existing coverage tool coverage.py on this function.

- **summary()**
  If the entry branch is counted, the number of branches is consistent with the coverage.py amount of branches. However, refactoring the ternary operators and for loops leads to an increased amount of branches as reported by coverage.py, in which case the difference increases.
- **concat()**
  Yes, branches reached and not reached by coverage.py match those of the tool.

In general, the tool is very accurate to the coverage reported by coverage.py, often within 5-10%. There are however cases where it differs by up to 20%, most likely due to our tool not counting lines within branches but simply the branches themselves.

Coverage improvement

Show the comments that describe the requirements for the coverage.

Function	Old Coverage	New Coverage	git diff
Convert_to_inference_data()	53%	69%	git diff e5575b 8b5fafe
plot_trace()	87%	93%	git diff 73ea437 bada5cb
plot_separation()	49%	91%	git diff e5575b585df36fe55076bf15fb8debfa0bbd5635 597981641f980630348c639a5ec74050e5f722e3
summary()	80%	93%	git diff 9eb7505 6286592
concat()	74%	81%	git diff 597981641f980630348c639a5ec74050e5f722e3 5b557cbcc5284a5f67be9840350fe696565fe88a

Coverage improvement convert_to_inference_data ()

Adding new tests was fairly simple since the function is directly callable, although the reliance on object types from other modules and packages made it fairly limiting in regards to what was reasonable to do. Many branches that are not covered would require fairly extensive reading into other packages in order to figure out how to generate data in their specific file format, and how to generate identical data in another format as a validation.

In the end, two test cases were added to check for the validity of the input. The first one sends a coords value along with an InferenceData object while the second one sends a dims value along with an InferenceData object, both of which are not allowed. The other tests send a csv file path as an object and verify that the conversion function is subsequently called correctly. A csv file may be sent with one of two values for the ‘group’ parameter, both which are tested along with not sending a group value at all.

Coverage improvement plot_trace()

The tests for plot_trace() are parameterized, meaning that adding new tests is very straight forward, as you can just add additional arguments to be run.

5 new test cases were added, each reaching a previously unreached branch. These branches are:
line 107: if not compact:
line 132: if isinstance(chain_prop, str):
line 135: if isinstance(chain_prop, tuple):
line 147: if isinstance(compact_prop, str):
line 150: if isinstance(compact_prop, tuple):

These new tests improve the coverage from 87% to 93%

Coverage improvement plot_seperation()

The new tests for the plot_seperation cover the ValueError exceptions that were not previously covered. The test cases basically do the same thing as they all assert that given a certain wrongful input a ValueError should be raised. To be specific, the test cases address these specific unreached branches

line 103: if idata is not None and not isinstance(idata, InferenceData):
line 107: if not all(isinstance(arg, (np.ndarray, xr.DataArray)) for arg in (y, y_hat))
line 117: elif y_hat is None:
line 122: elif not isinstance(y, (np.ndarray, xr.DataArray)):
line 128: elif not isinstance(y_hat, (np.ndarray, xr.DataArray)):

Coverage improvement summary()

The new tests for the summary cover the errors cases which were not covered. This includes invalid hdi_prob, kind, stat_focus and circ_var_names. I had intended to test the if not data.groups() branch but I could not find a way to create a data input without groups, since it does not seem that the setter function for the data object allows it.

Coverage improvement concat()

The new tests for concat cover some error cases that were not covered. It tests whether invalid dimensions, mismatch between the groups, and mismatch between the dimensions raise TypeError. Also, it tests whether common attributes are merged to a single value if both InferenceData objects contain common attributes. Also, ensures that different attributes remain unmerged.

Essence: Ways of Working

At the moment the group has reached the unanimous decision that we are in the “Foundation Established” state since we have not met all of the requirements listed in the “In Use” state. The following is an overview of our updated checklist for the “In Use” state:

- ☑ ~~The practices and tools are being used to do real work.~~
- ☑ ~~The use of the practices and tools selected are regularly inspected.~~
- ☑ ~~The practices and tools are being adapted to the team’s context.~~
- ☑ ~~The use of the practices and tools is supported by the team.~~
- ☐ Procedures are in place to handle feedback on the team’s way of working.
  - While the team is reflecting on our way of working, we do not at the moment have procedures in place for this. When someone notices an issue, it is brought up, but not in a structured or standardized way.
- ☑ ~~The practices and tools support team communication and collaboration.~~

The main obstacle we are currently facing in order to reach the next state is to establish practices/procedures that enable feedback on the team's way of working. This could for example be to establish weekly retrospective meetings where the team has the opportunity to review our way of working.

In our previous evaluation of our way of working we faced the same obstacle, so the team has seen no real improvement in this area. However the team has seen major improvements regarding the practices and tools to support team communication and collaboration, which was a previous obstacle. The team now agrees that we have sufficient practices and tools to enable effective communication and collaboration.

Overall Experience

One of the main take-aways from this project is that coverage tools are very useful for evaluating the code's robustness and quickly identifying weak spots. Another main takeaway is that setting up other peoples project can be quite a hassle if no proper documentation and troubleshooting is provided.

From this project the group has become more familiar with coverage tools and their strengths and limitations. The group has also learned the impact of applying the 'divide and conquer' technique for refactoring functions regarding reducing the branch coverage.