Group 24
Adam Sjöberg, Andreas Hammarstrand, Casper Kristiansson, Martin Lindefors, Victor Österman

# Assignment 3 - Code Coverage

## Onboarding

### Building the project

#### Required tools

There were no specific tools required to build the software. All that was required was to use either *pip* or *conda* for managing the developer environment.

#### Required Documentation

Both *pip* and *conda* are well established and thoroughly documented, hence the installation process was efficient and easy to follow.

#### Dependencies

There were 96 dependencies required to install *pytensor*. Both standard dependencies such as python and numpy, but also others. Additionally, to create a working developer environment there was a need to install *pytest* for testing, this information was not provided in the main README.

#### Build results

The build concluded with no errors.

#### Execution of examples and tests

First time we tried to run a code coverage tool, without correctly configuring the developer environment, we encountered issues running the tests. After building/installing the project by following the official documentation for *Pytensor*, there were no issues executing example code which implemented *pytensor*. As long as the developer follows the documentation for running tests, and has *pytest* installed, everything works as expected when executing tests. Important to note is that the documentation page associated with how to run the tests has a warning saying “This document is very outdated”. This might be a reason why we had initial issues running the code coverage tool.

#### Conclusion

Since the project is easy to build, works well with the correct requirements, and fulfills the project requirements from the assignment, the group has concluded to continue with PyTensor.

## Part 1: Complexity measurement

### Cyclomatic Complexity

For each of the five functions, we computed the CC in pairs of two. In each case, the pairs used the same CC equation, primarily since the results varied between the two methods. All pairs reached the same results for the five functions, with the exception of one pair where one member made a minor mistake in their computations. Most of the results did not entirely align with *lizard*. The following table contains our primary results for each of the five functions, with *lizard* results included:

| Function / CC Equation | $M = E - N + 2P$ | $M = \pi - s + 2$ | *Lizard* |
|---|---|---|---|
| Index_vars_to_types | Not used | $14$ | $24$ |
| Check_and_normalize_axes | Not used | $20$ (Excluding raise as exit point)<br><br>$18$ (Including raise as exit point) | $20$ |
| Belongs_to_set | Not used | $12$ | 18 |
| mean | $14$ | Not used | $19$ |
| get_canonical_form_slice | $15$ | Not used | 33 |

As visible from the table, our results diverged from the results of the *Lizard* tool. In all cases, we underestimated the CC in comparison to *Lizard*. We therefore decided to adapt our interpretation of the CC equation in the case of the function *check_and_normalize_axes*. If we only include the final *return* statement as an exit point, and ignore all *raise* statements, we achieve the same CC result as the tool. This is also the case in the *Belongs_to_set* function where, if we only interpret the last out of the total 7 return statements as an exit point, we get the same CC as *Lizard.* This shows how ambiguous CC computations seem to be since, depending on which equation and tool is used, different conclusions regarding the complexity will be reached.

### Correlation between high CC and LOC

| Function | LOC | CC |
|---|---|---|
| Index_vars_to_types | 63 | 24 |

| | | |
|---|---|---|
| Check_and_noramize_axes | 51 | 20 |
| Belongs_to_set | 79 | 18 |
| mean | 135 | 19 |
| get_canonical_form_slice | 177 | 33 |

Among the functions that the group has chosen, we can't see a pattern between more LOC and a higher CC, instead we see a pattern between decision branches and higher CC. In conclusion, in the functions that we have chosen, their density of decision branches is more related to the CC than the LOC.

### Function purpose and CC justification

index_vars_to_types:
This function takes an entry as input and converts references to Variables into references to Types. Since it handles different cases based on the type of entry and returns the corresponding Type or slice object it is related to high CC. It can be reduced by moving certain conditions into their own function and rewriting some structure of the code. For example, move isinstance instead of checking that several times.

check_and_normalize_axes:
This function transforms meta-data about axes to a python list of integers. This is done by evaluating the input variables, and depending on their type perform certain changes. The decision depends on several conditional statements, and the normalisation of the axis involves various loops, which consequently results in high CC. The high CC depends solely on the *axis* input variable. Depending on its type and dimensions, different branches will be executed. Since all these checks are necessary, the high CC in the function could be considered justified. However, to reduce the CC and possibly increase readability, it could be advantageous to create a helper function which evaluates the *axis* input variable. This would reduce the CC of the check_and_normalize_axes by moving some of the complexity to the helper function.

belongs_to_set: This function determines if a node belongs to a set of nodes in a sense that it can be merged together with every other node in the set. In order to be mergeable, they have to go over the same number of steps, have the same conditions (if there are any), have the same value for truncate gradient, and have the same mode. All the requirements mentioned above, are variables that each node contains when sent into the function. Since this function has to check many different things to be able to determine whether the node belongs to the set or not, it is necessary for the function to have a high CC. However, different parts and checks could be split up into helper functions to reduce the CC.

mean: This function computes the mean value of a given tensor input, i.e. a specific type of array. The mean depends on input variables which specify the behaviour required for the mean value. As a result, the function includes several conditional statements as well as

loops which lead to the high CC. Many of the loops and conditionals are to make sure that the data or format of the data is valid, and to transform it into a uniform data structure to process the actual mean on. As such, the high CC is justified, though some improvements are possible aside from separating it into more functions.

get_canonical_form_slice:
This function receives a Python slice as input and transforms it into a canonical form following the conventions of Python. It has to analyze the slice and its attributes. The function transforms a given slice [start:stop:step] into a canonical form. The canonical form ensures that: 0 <= start <= stop <= length, adjusting the start and stop values as necessary to fit within the bounds of the sequence length and to adhere to Python and NumPy conventions. Depending on the type of slice and the value of these attributes, different computations are performed. This behavior requires numerous conditional statements, which constitute all of the CC in the function. Another reason for the high CC for the function is that it handles many different cases. The function should be able to handle everything from normal slices to custom objects which means that there are a lot of edge cases to manage. The function also helps manage cases where the slice might be invalid and helps convert it.

### Python exception handling taken into consideration

The lizard tool takes try-except blocks into consideration when computing cyclomatic complexity. The try-statement is executed, and then there are two possible outcomes depending on whether or not an exception is raised. If an exception is raised, the except block is executed. If no exception is raised, the except block is ignored and the code execution continues. This results in two branches, which is why a try-except block increases the CC by one.

### Function documentation

One example of a function that successfully documents the possible outcomes by the different branches is the belongs_to_set function. The documentation describes that a node belongs to a set of nodes if it is mergeable with every other node in the set. The documentation then states what mergeable means. In other words, without describing every possible outcome in the code, the documentation clearly describes what each outcome would be. Conversely, the check_and_normalize_axes function contains no description of the various branches and outcomes possible in the function. However, since each of these are relatively simple, there is no need to describe all of them. The rest of the functions follow this pattern.

## Part 2

Task 1: DIY
The DIY coverage measurement tool is included within the tests and functions. Where we have made a list with false statements for each branch. Then in each function we change the False flag to a True flag if the branch is taken. After all the tests we have a teardown of the

test class that print the coverage % and the coverage list. Since we set the flags we cover both if, else, try, catch etc where a branch occurs. So the quality is based on how much we want to include as coverage and can be specific about what exactly we want to test. Our limitations of the tool is that you need to write the flags in each function which limit us that we can only check where we have written the flags; in case of a mistake or misunderstanding the tool will misrepresent the actual coverage. Furthermore, this means the tool is not automated and has to instrumented per function.

## Task 2: Coverage improvement

There is no simple way to determine and compare branch coverage in the project. We tried using automated tools, but they did not seem to provide accurate information. Each member of the group analysed the current test coverage of the analysed function and determined which requirements were covered, and which needed additional coverage. The next step was to implement tests which covered at least four of the untested branches. Each test was documented, with descriptions of the requirements it fulfilled/tested in the specific function. The assertions made were based on the members' understanding of the code, in combination with the documented purpose of the function and/or branch.

Each member followed the developer environment setup guide on the official website for *Pytensor*. Therefore the necessary modules were installed prior to running the tests. Each test called the function directly, and proceeded to make assertions on the returned results. Fortunately, we were not required to significantly change the structure of the testing suite or functions in order to implement and run tests.

Below is a table which shows the test branch coverage improvements made by our tests. These results are based on our own branch coverage detector.

| Function / Results | Coverage before(%) | Coverage after(%) |
|---|---|---|
| Index_vars_to_types | 35.29 | 70.59 |
| Check_and_normalize_axes | 0 | 50 |
| Belongs_to_set | 0 | 62.5 |
| mean | 40 | 55 |
| get_canonical_form_slice | 73.08 | 92.31 |

## Task 3: Refactoring plan

We decided to implement CC reduction for all functions, as each function contained some room for improvement. We describe each of the CC reductions below:

### index_vars_to_types:

The plan was to refactor some of the code by moving out a specific handle about slices where the code was moved out to their own function. Also a part that handles a specific error that could be a helping function was moved out. Another thing was to lump together the checks for Variables or Types since it was unnecessary to do that check several times and therefore it lowered the CC. By implemented thees things resulted in a reduction by 46%

### Belongs_to_set:

There are multiple branches in this function that have multiple branches. One of these is an if statement that contains two logical or operations, contributing to three decisions in one check. This check can be placed in a helper function that returns a boolean, which would reduce the amount of decisions on this line from three to one. Furthermore, there are a parsing step that parses the steps variables, which results in two try-except:s, by putting the parsing in another helper function, we can reduce the amount of decisions from two to zero. Lastly the function is creating two different nominal input lists, using for loops and if statements. By placing the creation of these lists in a helper function and return a tuple containing these two lists, the amount of decisions in belongs_to_set is reduced by four.

By implementing these helper functions, the CC, according to lizard, was reduced to 10, which resulted in a reduction by 44%.

### Check_and_normalize_axes:

This function had multiple purposes which perhaps should have been split up into several functions. It normalised the axes input by examining the type of the variable, and then proceeded to normalise its' values. If the results from this process were satisfactory, the function then validated the values of the normalised axes list according to certain conditions. To reduce the CC in this function, in addition to improving readability and maintaining modularised code, the plan was to create two utility functions which handled the normalisation and value validation separately. The most optimal solution would perhaps have been to create these two separate utility functions from the start, as normal functions, and call both of them instead of the check_and_normalize_axes. But since the purpose was to reduce CC, rather than completely eliminating it by removing the function, this was deemed the best course of action. The result was a 95% reduction of CC in the function.

### Get_canonical_form_slice:

The function had a lot of different logic that didn't have a lot to do with each other. For example, the function started by checking if the input even is a slice and if not it had logic handling that case and then returning a result for it. While if the input was a slice that code wouldn't be processed for it. That means that this logic could easily be extracted to its function. Simply extracting this logic gave a lot clearer image of what cases it was managing. As for the rest of the code it had first a step for pre-processing the slice and then a case where it checks if the slice starts with 0 or not. Each of these steps could be moved to its own function to help simplify the logic.

By implementing these helper functions, the CC, according to *lizard*, was reduced to 6, which resulted in a reduction of 82%.

### Mean:

The mean function contained a lot of transformations to a uniform data structure. Most of these transformations were common to any kind of statistical measurements or evaluations on matrices. This meant that some parts could be replaced by a utility function to handle the transformation, one that could later be merged or replaced by a generalised function for the same usage.

For example, a part of calculating the mean is determining on which axis or axes to calculate it on. Axis is an input argument that may also be None, which occurs in other functions like sum, and as such can use the same transform function for this behaviour. However, since they are not identical in their current format they were simply extracted from the function to allow easier replacement later.

## Task 4: Self-assesment

This assignment was very different from previous ones, making it more difficult to determine where and what we have accomplished on the checklist of the Essence standard. However, we bring along our previous experiences to this assignment, meaning we start off at around the same place as before; “In Use”. We have established many of the important principles of working with essence through plain practice and many of the important factors come naturally now. However, since this assignment was aimed more at a theoretical standpoint and a report, we could not live up completely to “In Place” as where should go where and when quickly became chaotic. We think that if this assignment had been aimed more towards the pure practical work of Open Source, we would have reached “In Place” and even worked on some parts of “Working Well”, though perhaps not enough to check them off in the checklist.