IOWR-NIDS

Software Testing Document

Date: 2/25/2024

Team:

Andrew Bajumpaa

Alexandre Broggi

Joseph O'Dowd

# Testing plan:

Our testing plan is to test the model and the model interfaces with unit tests but due to how the GUI works we will only be able to perform integration and acceptance tests for it. The sections titled “Test\_” are mainly unit tests and the sections titled “… Testing” are mainly implementation tests.

# Sections:

* Test\_HelperFunctions – Tests the helperfunctions.py file with unit tests.
* Test\_LoadModelData – Tests if the data can be loaded for model training with unit tests.
* Test\_LoadModelRun – Tests if the model runs properly and can be loaded from a file with unit tests.
* Dataloader Testing – Implementation tests of the data pipeline from wireshark data collection.
* Interface Testing – Tests if the interface loads correctly and if it has the correct information.
* Start script Testing – Tests if the starting shell script works properly.

# Test\_HelperFunctions

Test neural network model functions with low cohesion with the rest of the model training process.

All these tests are run every update.

## Test\_seenModels

* + Test type: Unit Test
  + What is being tested:
    1. Testing function to read filenames of already created models. We should be able to collect all the file names so we can load up the models.
  + Test Procedure:
    1. Pytest unit test
    2. Run function from code and collect array.
    3. Iterate through array and assert that the files exist.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

All files found by the function do exist.

## Test\_loopOverUnknowns

* + Test type: Unit Test
  + What is being tested:
    1. Changes to Config for training with new classes. When the classes are changed due to adding new data, we should see the classes change in the Model Config to reflect the new possible classes.
  + Test Procedure:
    1. Save current model identifiable classes.
    2. Assert that Unknowns and Knowns are different.
    3. Run the function “loopOverUnknowns”.
    4. Assert that new Unknowns and Knowns are different from each other.
    5. Assert that the new classes were added.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Adding classes works on the level of loading the data.

# Test\_LoadModelData

Test data loading for the model training process. Mostly checking if the data is in the correct format and does not have illegal values.

All these tests are run every update.

## test\_checkAttemptLoad

* + Test type: Unit Test
  + What is being tested:
    1. Dataset instances for training, testing, and validation are loaded properly from files.
  + Test Procedure:
    1. Load datasets with checkAttempLoad.
    2. Assert that each dataset is an instance of “Dataload.Dataset".
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Data can be loaded from a savepoint.

## test\_createLoaders

* + Test type: Unit Test
  + What is being tested:
    1. Test that each of the datasets loaded from “checkAttempLoad” can be properly converted into iterators and that the output is in the correct format.
  + Test Procedure:
    1. Set the dataset to the unit testing dataset that has fewer samples.
    2. Use checkAttempLoad to load the datasets, training, testing, and validation.
    3. Create iterators for each dataset.
    4. Get each batch from the iterators.
    5. Assert that the batch is made of a tuple of two tensors.
    6. Assert that the target tensor is the correct shape.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Dataloaders work properly and output the correct datatypes.

## test\_validation\_dataset

* + Test type: Unit Test
  + What is being tested:
    1. Validation dataset only contains data that is meant to be trained on.
  + Test Procedure:
    1. Set the dataset to the unit testing dataset that has fewer samples.
    2. Use checkAttempLoad to load the datasets, training, testing, and validation.
    3. Create iterator for Validation dataset.
    4. Assert that all data in each batch is only from the known classes.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Validation dataset is not contaminated with unknown data, which would invalidate the validation dataset.

## test\_testing\_dataset

* + Test type: Unit Test
  + What is being tested:
    1. Only Unknown data is in the testing dataset when Config is set to require it.
  + Test Procedure:
    1. Set the dataset to the unit testing dataset that has fewer samples.
    2. Set “Mix unknowns and validation” to 0 so that only unknown samples will be used to build the testing dataset.
    3. Check that Config is not set to a value that will break the test, if true, skip the test.
    4. Use checkAttempLoad to load the datasets, training, testing, and validation.
    5. Create iterator for Testing dataset.
    6. Assert that all data in each batch is only from the Unknown classes.
  + Results (or Pass/Fail for unit tests):
    1. Skipped at step 3,

Model is currently not set to have any unknowns, so this test is currently not being applied.

## test\_add\_new\_class

* + Test type: Unit Test
  + What is being tested:
    1. If Config is able to be updated in the event that a new class gets added to the list of classes.
  + Test Procedure:
    1. Add new dummy class name to list of classes.
    2. Add new dummy class to the inverted list of classes.
    3. Collect the current number of classes.
    4. Update the number of classes.
    5. Assert that the new number of classes is not the same as the old number of classes.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

New classes are added correctly to the Config so that we can train on new data.

## test\_add\_new\_class\_function

* + Test type: Unit Test
  + What is being tested:
    1. Test that a function designed to add new classes to the classlists works properly. Classlists are dictionaries that translate the class names into integers and back for model training and identification.
  + Test Procedure:
    1. Collect the current number of items in classlist and listclass.
    2. Add a new dummy class with “add\_new\_class”.
    3. Assert that the new number of items in classlist and listclass are not equal to the old numbers.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

New classes are added correctly to the CLASSLIST so that we can train on new data.

## test\_dynamic\_dataloader

* + Test type: Unit Test
  + What is being tested:
    1. That the dynamic data loader outputs data in the correct format for model training.
  + Test Procedure:
    1. Create a dynamic data loader.
    2. Initialize an iterator for the dynamic data loader.
    3. For each batch in the iterator assert the following:
       1. The true target column and the training target column are equal.
       2. The true target column is within bounds of possible classes.
       3. The length of the training data is the same as the rest of the data (1504 columns).
       4. The first item in the targets column is a valid Class.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Test that the dynamic data loader uses the add classes function properly and that it is in the correct format.

# Test\_LoadModelRun

Test if the model runs correctly and outputs data in the correct format. These are a bit more of implementation tests then unit tests.

All these tests are run every update.

## test\_modelReads

* + Test type: Unit Test
  + What is being tested:
    1. Tests that the model is outputting a torch tensor.
  + Test Procedure:
    1. Set Dataset to unit test.
    2. Initialize and retrieve a batch from the training dataset.
    3. Run the dataset through the model.
    4. Test if output is a torch tensor.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Model reads in the data properly, further tests can be conducted.

## test\_modelReads\_labeledUnknowns

* + Test type: Unit Test
  + What is being tested:
    1. Test that the saved packet dataset is able to be read into the model and produce results.
    2. Note: this does not test if the model will be able to predict any of the classes from the new dataset only that the dataset is in the correct format.
  + Test Procedure:
    1. Initialize a “savedPacketDataset” iterator.
    2. Get a batch from the iterator.
    3. Pass that batch through the model.
    4. Test if the model outputs a tensor.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The model is able to read the new datasets.

## test\_modelTrain

* + Test type: Implementation Test
  + What is being tested:
    1. If the model’s validation loss decreases at all over a short training time.
    2. This verifies that the backpropagation is working and that the model is training in the correct direction.
  + Test Procedure:
    1. Get a model.
    2. Pass the training data through the model evaluation method.
    3. Train the model for six epochs at a low learning rate (to avoid loss increases)
    4. Pass the training data through the model again.
    5. Check that the validation loss decreased.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

No errors, so that means that the model is able to improve, even if it is just a slight improvement.

## test\_modelDataObject

* + Test type: Unit Test
  + What is being tested:
    1. This tests that the model is able to generate valid output objects for the dataload section to read.
  + Test Procedure:
    1. Get a batch from the training dataset instance created earlier.
    2. Train the model for 1 epoch quickly just to make sure that all internal values are set.
    3. Call “generateDataObject” on the batch from the training dataset.
    4. Verify that the number of attack classes in the data object are equal to one less than the number of total classes from config.
    5. Make sure that the number of packets is identical to the length of the batch.
    6. Check that the unknowns are a list.
    7. Check that the predictions, predictions\_numerical, and predictionProbability each have an associated packet by checking the length.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The data object can be passed to the GUI.

## test\_loadModelOld

* + Test type: Integration Test
  + What is being tested:
    1. Old method of detecting that the model loading method works properly.
    2. This test is finicky, and we only keep it around because tests are supposed to find things that are broken, it does not take much to trigger this test.
  + Test Procedure:
    1. Get saved models from helperfunctions.
    2. Check that we have more than two models saved, if not, train six epochs and save after each epoch. Also refresh the list of saved models.
    3. Load model at index 0 from list.
    4. Pass a batch through that model.
    5. Load model at index 1 from the list
    6. Pass the same batch through that model.
    7. Load the model from index 0 again.
    8. Pass the same batch through the model.
    9. Verify that the first and the third outputs were equal.
    10. Verify that at least one value between the first and the second outputs were not equal.
    11. If that last test was not true, try loading a different model and replacing the second output with one from the new model. Repeat this until you run out of models, or you find one that is different (because there is no guarantee that every model will have different outputs but at least one of them should.)
    12. If you ran out of models to test, Fail the test.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

There exist two saves that have different models.

## test\_loadModel

* + Test type: Implementation Test
  + What is being tested:
    1. Test that the model nodes are different before and after a load.
    2. That is, that loads change the model node values (they work)
  + Test Procedure:
    1. Get saved models from helperfunctions.
    2. Check that we have more than two models saved, if not, train six epochs and save after each epoch. Also refresh the list of saved models.
    3. Create two models.
    4. Load a different index from the model names into each model.
    5. Go through every parameter in the models and compare them.
    6. If every parameter is equal, Fail the test.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The load function works, and the model changes its parameters when a load is attempted.

## test\_model\_quicktrain

* + Test type: Implementation Test
  + What is being tested:
    1. Test that the quick training method changes the model node values.
  + Test Procedure:
    1. initialize two models with the same randomizer seed so that they are equal.
    2. Train the second model with “train\_model”
    3. Verify that not every parameter in both models is the same.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The training function changes the model.

## test\_model\_dates

* + Test type: Implementation Test
  + What is being tested:
    1. Test that the model saves include a date of creation.
  + Test Procedure:
    1. Load a model from the model list.
    2. Check that it has a date in the date format.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Loaded model has a date of creation attribute associated with the model.

## test\_model\_info

* + Test type: Implementation Test
  + What is being tested:
    1. Test that all model saves are able to be loaded with the model info function.
  + Test Procedure:
    1. Load a model info from the model list that you have not loaded yet.
    2. Check that has each of these fields: params, classes used, confusion matrix, first created, last updated, and benign f1-score.
    3. Check that each of those is of the correct type.
    4. Go back to step 1 until there are no more models to test.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

All model attributes are able to be loaded on each model info call, so we can get the info of any model.

# DataLoader Testing

A series of implementation tests designed to validate the DataLoader’s ability to complete the network-database-model-database-interface pipeline.

## networkFeed thread test

* + Test type: Integration Test
  + What is being tested:
    1. The networkFeed thread’s sufficiency to send batches of packets to the DataLoader for further processing.
  + Test Procedure:
    1. Enable networkFeed’s start within DataLoader.
    2. If on a new system, change networkFeed’s interface to the main network adapter.
    3. Run DataLoader.
    4. Verify that batches are being received by the DataLoader with print statements.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The networkFeed currently passes the test, feeding dataframes of 100 packets into the DataLoader. The only issue it currently faces is input blocking.

## Model loading test

* + Test type: Integration Test
  + What is being tested:
    1. DataLoader’s ability to load models per-client for the processing of packets.
  + Test Procedure:
    1. Start DataLoader and Client.
    2. Open the client in a browser.
    3. Verify that the model list populates.
    4. Using the DataLoader’s output, verify that the model has loaded without critical failure.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The DataLoader successfully signals that models have loaded, and packets from networkFeed are processed properly. An optional debug print can be used to look at the results before they enter the database.

## Database connectivity test

* + Test type: Integration Test
  + What is being tested:
    1. The connection to PostgreSQL.
  + Test Procedure:
    1. Update the PostgreSQL connection information if necessary.
    2. Start DataLoader.
    3. Verify that there is no error about the connection failing (a connection attempt is made immediately).
  + Results (or Pass/Fail for unit tests):
    1. Pass,

No error prints, indicating success. The database can successfully be operated on with queries.

## Full packet pipeline

* + Test type: Integration Test
  + What is being tested:
    1. networkFeed -> Packet DB -> model -> pack\_class DB pipeline
    2. The full input pipeline for inserting packets into the database.
  + Test Procedure:
    1. Start DataLoader.
    2. Load a model with the client.
    3. Verify that batch, packet DB insertion, and pack\_class DB insertion messages are being output. There will be verbose errors if not.
    4. Use the Dash client (or, if that’s not working, a SQL client) to verify the program did what it said it did.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

After starting DataLoader, the networkFeed indicates that it is sending batches. DataLoader indicates that the packets are sent to the database. The model then creates labels; these are uploaded to a separate database table. Populated tables are visible within Pgadmin or the Dash interface.

## Client connectivity test

* + Test type: Integration Test
  + What is being tested:
    1. The client’s ability to interface with DataLoader and persist data between sessions.
  + Test Procedure:
    1. Start DataLoader and Client.
    2. Load a model from within the client’s model dropdown.
    3. Look for information about the client’s model being loaded within the DataLoader’s output.
    4. In the same way, verify that DataLoader does not load the model again upon the page reloading. The DataLoader should say that the client is merely connecting to the model.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The client’s CLI output indicates a successful connection, and DataLoader indicates that it is connecting to a model instead of loading new ones each time the same machine connects.

## Client data loading

* + Test type: Integration Test
  + What is being tested:
    1. Packet DB JOIN pack\_class DB -> Client pipeline
    2. The ability for the client to retrieve information from DataLoader.
  + Test Procedure:
    1. Start DataLoader and Client.
    2. Verify that the model dropdown, packet table, and graphs are immediately fully populated.
    3. Verify that the packet table’s sorting and filter functions work.
    4. Click on a packet in the table to verify that individual packet data can be read.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

The webpage successfully populates with tables and visuals based on queries to the database.

# Interface Testing

A series of implementation tests to verify the interface’s functionality.

## Dash testing

* + Test type: Integration Test
  + What is being tested:
    1. The ability to spin up a Dash server.
  + Test Procedure:
    1. Start Client.
    2. Verify that it outputs a link with a status update (likely <http://localhost:8050>).
    3. Visit the link, verifying that the page loads without stalling.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Page should load in, connect to server, and display visuals.

## Callback testing

* + Test type: End to End Test
  + What is being tested:
    1. The callbacks that share a few inputs.
  + Test Procedure:
    1. Start DataLoader and Client.
    2. Visit the web client.
    3. Verify that inputs change data or at least print status updates in Client’s output.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Data passes between client and backend correctly.

## Screen testing

* + Test type: End to End Test
  + What is being tested:
    1. The web app’s conformity to screens.
  + Test Procedure:
    1. Start DataLoader and Client.
    2. Visit the web client.
    3. Verify that the content does not stretch outside of the page nor clip itself.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

Page looks good and displays information in a readable format.

## Usage testing

* + Test type: Acceptance test
  + What is being tested:
    1. Whether it meets expectations of the client and end users
  + Test Procedure:
    1. Start DataLoader and Client.
    2. Visit the web client.
    3. Gather usage data and user feedback.
  + Results (or Pass/Fail for unit tests):
    1. Pass,

We have received positive and constructive feedback.

# Start Script Testing

Tests to identify if the script made to start the program works without major errors.

## Script launch test

* + Test type: Integration Test
  + What is being tested:
    1. The script, “start.sh”, runs without an error.
  + Test Procedure:
    1. Start a terminal window.
    2. Start the Python venv.
    3. Run “start.sh” in the terminal window.
    4. Check that no errors appear in the console.
  + Results (or Pass/Fail for unit tests):
    1. Pass,  
       No errors appeared in the console.

## Script client window test

* + Test type: Integration Test
  + What is being tested:
    1. The script, “start.sh”, opens the client window.
  + Test Procedure:
    1. Start a terminal window.
    2. Start the Python venv.
    3. Run “start.sh” in the terminal window.
    4. Check that the window appears after launch.
  + Results (or Pass/Fail for unit tests):
    1. Pass,  
       Window in safari opened as expected.

## Script stop test

* + Test type: Integration Test
  + What is being tested:
    1. The script, “start.sh”, closes with “control+c”.
  + Test Procedure:
    1. Start a terminal window.
    2. Start the Python venv.
    3. Run “start.sh” in the terminal window.
    4. Press “control+c”, “terminated ./start.sh” should appear.
    5. Verify in Activity Monitor/ Task manager that no large Python programs are running after termination.
  + Results (or Pass/Fail for unit tests):
    1. Pass,  
       No Python processes are using CPU after termination.