MCS23 FIT3162 Software Test/QA Report

Team: MCS23
Project Title: Information-based associative analysis and deep learning for classifying time-series data

1. Introduction

1.1 Introduction

Time series data is present everywhere in today's world, appearing in various fields such as economics, healthcare and science. Therefore, harnessing the power to predict time series data is essential so that important decisions and effective strategies can be developed beforehand.

Our team has developed a web prototype that uses Association Rule Mining (ARM) in combination with a Long short-term memory (LSTM) machine learning model that is able to perform prediction on time series data. To ensure that our prototype meets the product requirements and is bug free, we tested our prototype extensively. Through testing, we are also able to gain informative feedback from users and further improve our prototype.

1.2 Test approach and plan

To achieve our testing objectives, different testing methods are used to test different parts of the prototype. Automated te-sting methods are mostly used to test the backend of our prototype, including the source code for data preprocessing, ARM and LSTM. To perform automated testing, the python built in module unittest which provides a unit testing framework is used. The unittest module allows us to write test scripts to test different parts of the source code. Manual testing is mainly used to test the frontend of the prototype, including evaluating inte-ractions with the user interface-, ensuring the integrity of visualisations, and validating the- accuracy of results. Additionally, pe-rformance testing will assess the- response time and e-fficiency of the prototype. Usability tests will be done by collecting and analysing feedback from test use-rs. Once bugs or areas of improvement are uncovered through the tests, code modifications will be made and the related tests will be repeated.

Our team will first focus on testing the functional requirements then the nonfunctional requirements. For every test, the tester name, requirements covered, test method, test description, area tested, expected output and actual output are recorded. The test plan used is the same as the one in our team proposal (see Appendix F Table 2) but without the tests that test the removed requirements. These tests are divided equally among all 3 members, with members testing on parts they are more familiar with.

1.3 Product Requirements

Below are the product functional and nonfunctional requirements that will be tested.
Functional requirements:
FR1 - Allows for data upload: Users are free to upload any data that they want to predict.
FR2 - Preprocesses the data: This will improve the prediction accuracy.
FR3 - Implements Apriori algorithm for association rule mining: To identify patterns and relationships between variables in the data.
FR4 - Implements deep learning architecture LSTM: Suitable deep learning architecture for predicting time-series data.
FR5 - Able to produce prediction results: The product can achieve its purpose of predicting future time-series data points.
FR6 - Can generate graphs and charts that describe the prediction results: Can visualise prediction alongside actual values for comparison.
FR7 - Implements metrics to measure performance of the model: Use reliable ways and formulas to evaluate product performance.

Nonfunctional requirements:
NFR1 - Run time for prediction is within acceptable time limit (5 seconds): Ensure short response time.
NFR2 - The prediction accuracy exceeds a certain threshold: Ensure product is trustworthy to do decision making.
NFR3 - Recovers gracefully from errors: Prevent crashing of system and loss of data.
NFR4 - Always running, with minimal downtime for maintenance: Product can be used most of the time
NFR5 - User friendly, has a simple design and is easy to navigate: Can be easily used without understanding how the backend system works.
NFR6 - Documentations and comments included in code: Easy to understand how the written codes work.

Some of the requirements were changed from the proposal with reasons as shown below.
Functional requirements:
Accepts data of various formats including csv, json and others: Do not have enough time to implement, therefore only meet minimum requirement to accept data of type .txt
Extracts only the rules that exceed the threshold: Is included in FR3 where Apriori will automatically discard rules below threshold
Supports hyperparameter tuning: Too complex for users to understand how to perform
Provides a detailed report on its performance: Performance already shown through FR6 and FR7

Nonfunctional requirements:
Can process large amounts of time-series data: Dependent on computer specs
Implements user authentication mechanisms: Data is deleted after user stops using the product, therefore no need to store data
User data is encrypted and stored securely in the database: Data is deleted after user stops using the product, therefore no need to store data
Designed with maintainable code: Do not have enough time to design code in a maintainable way

2. Whitebox Testing

Whitebox Test 1

Tester name: David Lee
Requirements covered: FR3
Test description: In this test, the application of ARM technique apriori algorithm is tested. Since ARM is applied during the creation of Data_util object where the feature selection is applied to the data, the test is done by checking if ARM runs successfully and the number of rows of data after ARM are less than before ARM.

```
def test_arm(self):
    print("Test arm")
    file1 = open('data/electricity.txt')
    file2 = open('data/solar_AL.txt')
    no_rows1 = len(np.loadtxt(file1, delimiter=',')[0])
    no_rows2 = len(np.loadtxt(file2, delimiter=',')[0])
    file1.close()
    file2.close()

    self.assertLess(len(self.data1.rawdat[0]), no_rows1)
    self.assertLess(len(self.data2.rawdat[0]), no_rows2)
```

Part of code tested: def _arm of class Data_util
Expected output: ARM runs successfully and number of rows of data after ARM are less than before ARM
Actual output: ARM does not run successfully

```
Error
Traceback (most recent call last):
  File "C:\Users\Asus\Documents\Homework\FIT3162\LSTM\TestData.py", line 9, in setUp
    self.data2 = LSTM_train_eva.Data_util('data/solar_AL.txt', 0.7, 0.2, False, 12, 24 * 7, 0)
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\Documents\Homework\FIT3162\LSTM\LSTM_train_eva.py", line 56, in __init__
    self._arm(5, 0.7, 0.7, 50)
  File "C:\Users\Asus\Documents\Homework\FIT3162\LSTM\LSTM_train_eva.py", line 91, in _arm
    rules = association_rules(frequent_itemsets, metric='confidence', min_threshold=min_threshold)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\Documents\Homework\FIT3162\venv\Lib\site-packages\mlxtend\frequent_patterns\association_rules.py", line 83, in associa
    raise ValueError(
ValueError: The input DataFrame `df` containing the frequent itemsets is empty.
```

Whitebox Test 2

Tester name: David Lee
Requirements covered: FR3
Test description: In this test, the application of ARM technique apriori algorithm is tested. This test is a repeat test after code modification of the ARM is done. The code is modified so that when no rules are generated by ARM, feature selection is not done on the data. The test is done by checking if ARM runs successfully and the number of rows of data after ARM are less than or equal to before ARM.

```
def test_arm(self):
    print("Test arm")
    file1 = open('data/electricity.txt')
    file2 = open('data/solar_AL.txt')
    no_rows1 = len(np.loadtxt(file1, delimiter=',')[0])
    no_rows2 = len(np.loadtxt(file2, delimiter=',')[0])
    file1.close()
    file2.close()

    self.assertLess(len(self.data1.rawdat[0]), no_rows1)
    self.assertEqual(len(self.data2.rawdat[0]), no_rows2)
```

Part of code tested: def _arm of class Data_util

Expected output: ARM runs successfully and number of rows of data after ARM are less than or equal to before ARM
Actual output: ARM runs successfully and number of rows of data after ARM are less than or equal to before ARM

```
Test arm


Ran 1 test in 23.184s

OK
```

Whitebox Test 3

Tester name: David Lee
Requirements covered: FR2
Test description: In this test, the splitting of data into train, validate and test data are tested. The test is done by checking if the data are split according to the given ratio.

```
def test_split(self):
    print("Test split")
    file1 = open('data/electricity.txt')
    file2 = open('data/solar_AL.txt')
    data1 = np.loadtxt(file1, delimiter=',')
    data2 = np.loadtxt(file2, delimiter=',')
    file1.close()
    file2.close()

    no_train_cols1 = int(len(data1) * 0.6) + 1 - 24 * 7 - 12
    no_val_cols1 = int(len(data1) * 0.2) + 1
    no_test_cols1 = int(len(data1) * 0.2) + 1

    no_train_cols2 = int(len(data2) * 0.7) + 1 - 24 * 7 - 12
    no_val_cols2 = int(len(data2) * 0.2) - 1
    no_test_cols2 = int(len(data2) * 0.1) + 1

    self.assertEqual(len(self.data1.train[0]), no_train_cols1)
    self.assertEqual(len(self.data1.valid[0]), no_val_cols1)
    self.assertEqual(len(self.data1.test[0]), no_test_cols1)

    self.assertEqual(len(self.data2.train[0]), no_train_cols2)
    self.assertEqual(len(self.data2.valid[0]), no_val_cols2)
    self.assertEqual(len(self.data2.test[0]), no_test_cols2)
```

Part of code tested: def _split of class Data_util
Expected output: Data are split according to the given ratio
Actual output: Data are split according to the given ratio

```
Ran 1 test in 40.739s

OK

Process finished with exit code 0
Test split
```

White box test 4

Tester name: Liang Dizhen

Requirements covered: FR4, NFR1, 2, 4

Test description: In this test, the LSTM model is implemented for testing whether it can be trained or not by asserting whether its parameters have been changed or not.

```
class TestModelTraining(unittest.TestCase):

  def setUp(self):
      # Initialize a Data_util object with mock data and parameters
      self.data = LSTM_train_eva.Data_util('data/electricity.txt', 0.6, 0.2, False, 12, 24 * 7, 2)
      self.X_train, self.Y_train = self.data.train[0], self.data.train[1]# Get training data batches

      # Initialize a simple LSTM model
      self.model = LSTM_train_eva.LSTM(input_size=len(self.X_train[0][0]), hidden_size=128)

      # Initialize a loss function and optimizer
      self.criterion = torch.nn.MSELoss()
      self.optimizer = torch.optim.Adam(self.model.parameters(), lr=0.001)

      # Move model to GPU if available
      self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
      self.model.to(self.device)
      self.X_train = self.X_train.to(self.device)
      self.Y_train = self.Y_train.to(self.device)
```

```
  def test_model_weights_update(self):
    def train(model, data, X, Y, criterion, optimizer, batch_size):
      model.train()  # Set the model to training mode
      total_loss = 0
      n_samples = 0
      count = 0

      for X_batch, Y_batch in data.get_batches(X, Y, batch_size, True):
        #run once only
        if count < 3:
          optimizer.zero_grad()  # Clear gradients
          hidden_state, output = model(128,X_batch)  # Forward pass
          loss = criterion(output, Y_batch)  # Compute loss
          loss.backward()  # Backward pass
          optimizer.step()  # Update weights

          total_loss += loss.item()
          n_samples += X_batch.size(0)  # Increment sample count
          count += 1
      return total_loss / n_samples

    # Before training, save the initial weights
    initial_weights = [param.data.clone() for param in self.model.parameters()]

    # Perform one training step
    train(self.model, self.data, self.X_train, self.Y_train, self.criterion, self.optimizer, batch_size=128)

    # After training, check if the weights have changed
    updated_weights = [param.data.clone() for param in self.model.parameters()]

    # Check if all corresponding weights in the lists are equal
    weights_equal = all(torch.equal(iw, uw) for iw, uw in zip(initial_weights, updated_weights))

    self.assertFalse(weights_equal, "Weights should be updated during training")
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Part of code tested: class LSTM

Expected output: LSTM model is trained and its parameters have changed

Actual output: LSTM model is trained and its parameters have changed

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future.
  and should_run_async(code)
.
----------------------------------------------------------------------
Ran 1 test in 5.191s

OK
```

3. Blackbox Testing

Blackbox Test 1

Tester name: David Lee
Requirements covered: FR1
Test description: In this test, the success of the creation of the Data_util object is tested. The test is done by creating Data_util objects using different data and checking if the objects successfully create without errors and are of class Data_util.

```
def setUp(self):
    self.data1 = LSTM_train_eva.Data_util('data/electricity.txt', 0.6, 0.2, False, 12, 24 * 7, 2)
    self.data2 = LSTM_train_eva.Data_util('data/solar_AL.txt', 0.7, 0.2, False, 12, 24 * 7, 0)

def test_data_object_creation(self):
    print("Test data object creation")
    self.assertIsInstance(self.data1, LSTM_train_eva.Data_util)
    self.assertIsInstance(self.data2, LSTM_train_eva.Data_util)
```

Part of code tested: class Data_util
Expected output: Data_util objects successfully create without errors and are of class Data_util
Actual output: Data_util objects successfully create without errors and are of class Data_util

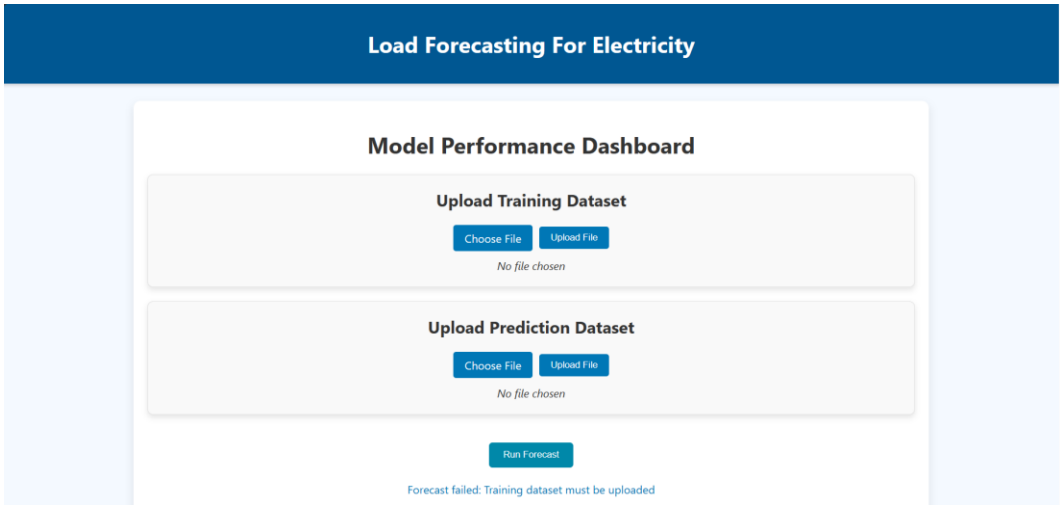
```
Test data object creation

Ran 1 test in 47.773s

OK
```

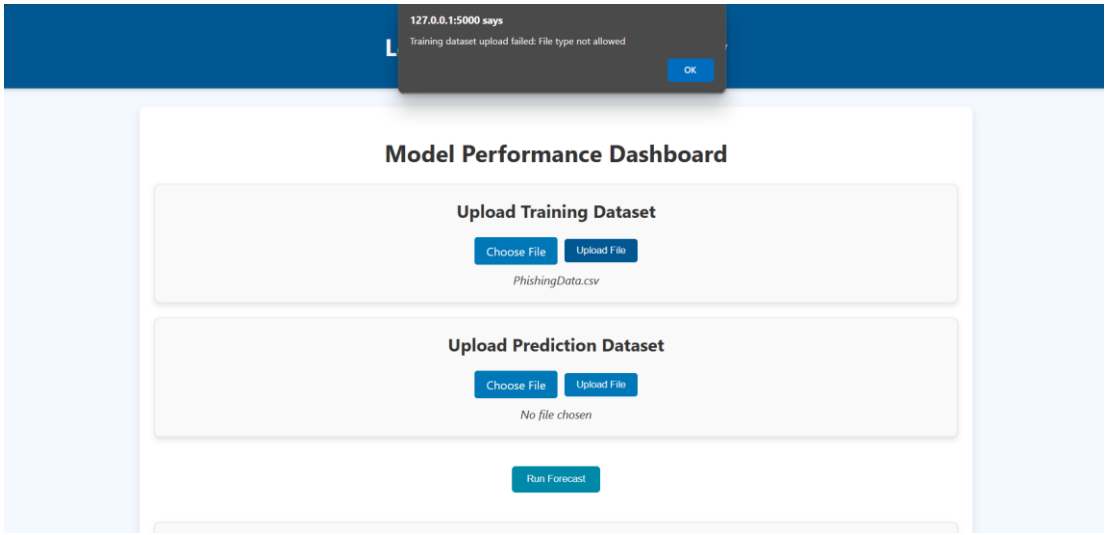
Blackbox Test 2

Tester name: David Lee
Requirements covered: NFR3
Test description: Test that the prototype can handle the error of running the forecast without any data being uploaded. This test is conducted by not choosing any files to upload and pressing the “Run Forecast” button.
Part tested: Frontend of prototype
Expected output: Error message stating reason of error shown and prototype does not crash
Actual output: Error message stating reason of error shown and prototype does not crash



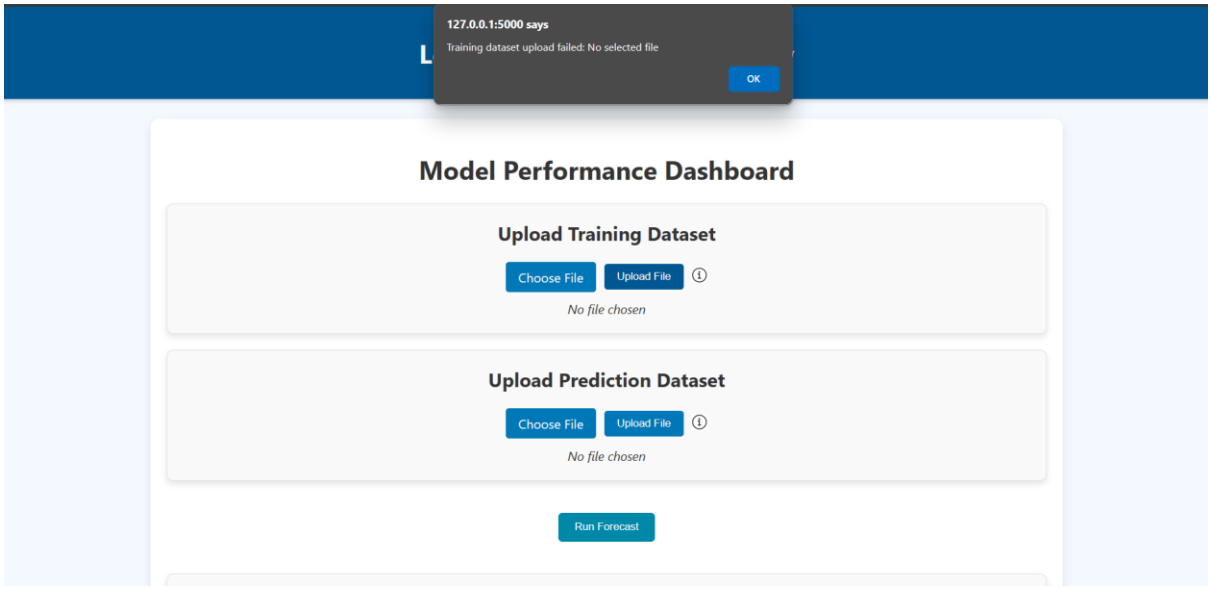
Blackbox Test 3

Tester name: David Lee
Requirements covered: FR1, NFR3
Test description: Test that the prototype can handle the error of uploading data of incorrect format. This test is conducted by choosing a data file of type csv and pressing the “Upload File” button.
Part tested: Frontend of prototype
Expected output: Error message stating reason of error shown and prototype does not crash
Actual output: Error message stating reason of error shown and prototype does not crash



Blackbox Test 4

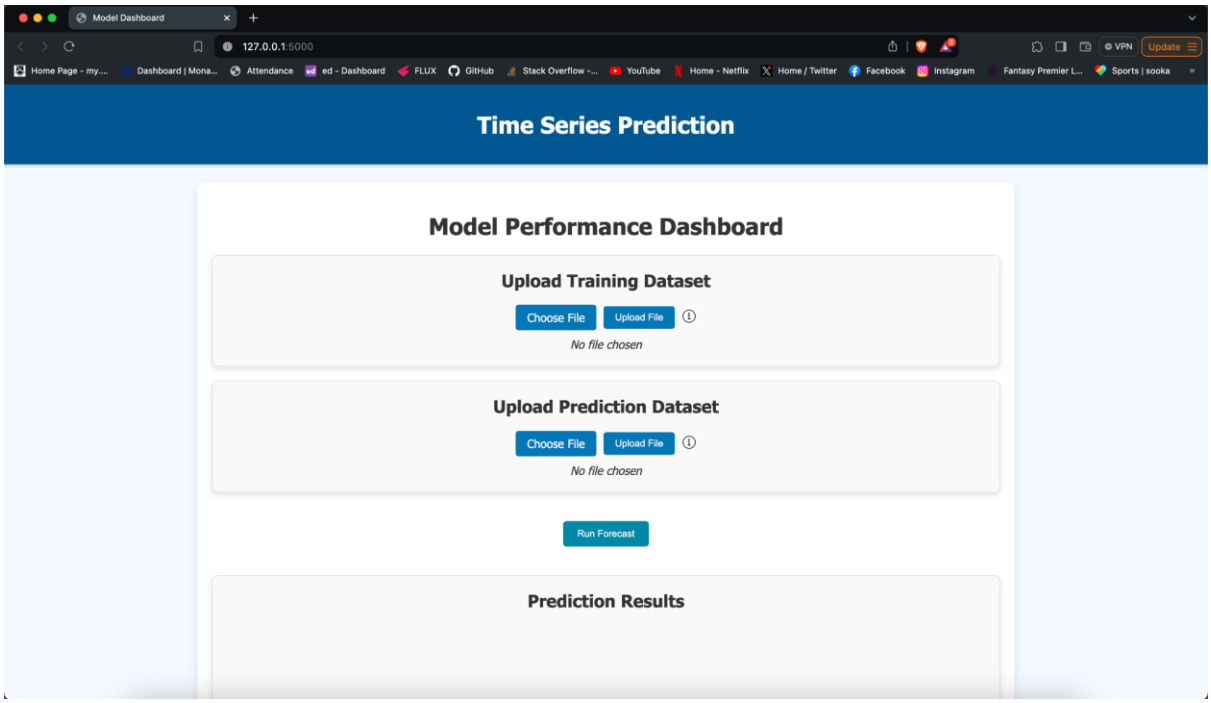
Tester name: David Lee
Requirements covered: FR1, NFR3
Test description: Test that the prototype can handle the error of uploading data without choosing any file. This test is conducted by not choosing any file any press the “Upload File” button.
Part tested: Frontend of prototype
Expected output: Error message stating reason of error shown and prototype does not crash
Actual output: Error message stating reason of error shown and prototype does not crash



4. Integration Testing

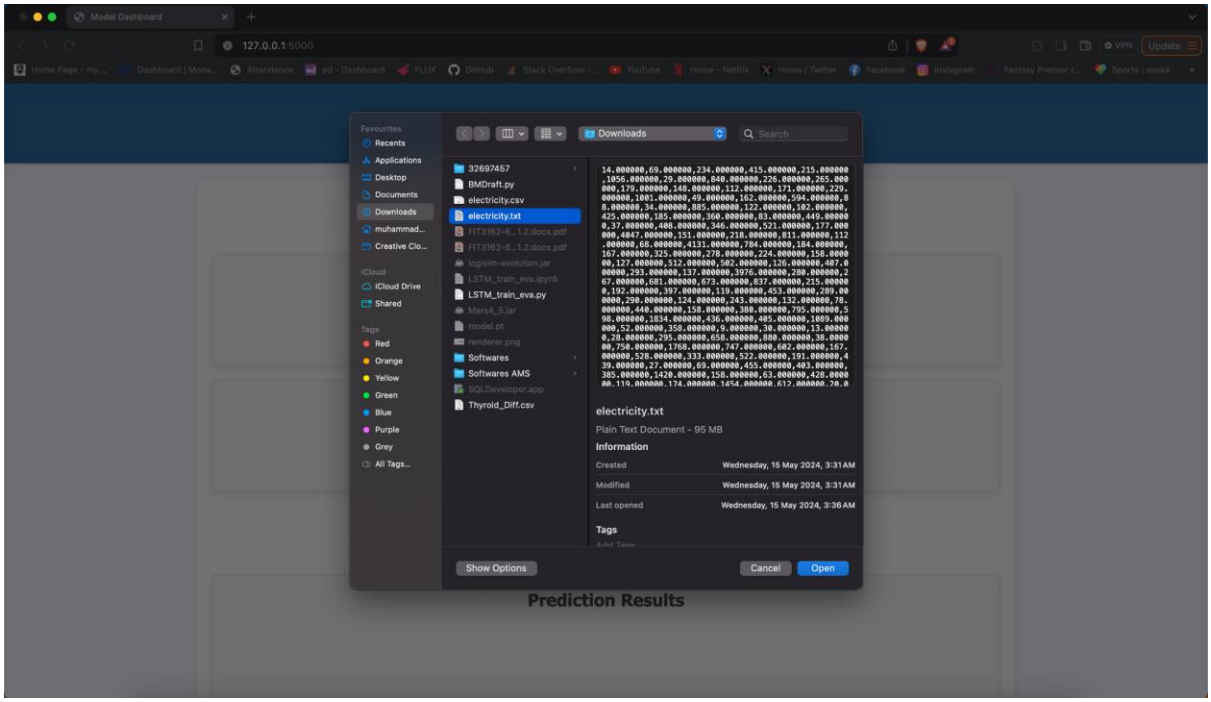
Each screenshot demonstrates a key step in the process of uploading datasets, running the forecast, and visualising the prediction results, ensuring that all functionalities work as expected.

Screenshot 1:
Tester name: Muhammad Abdullah Akif
Requirements covered: NFR5



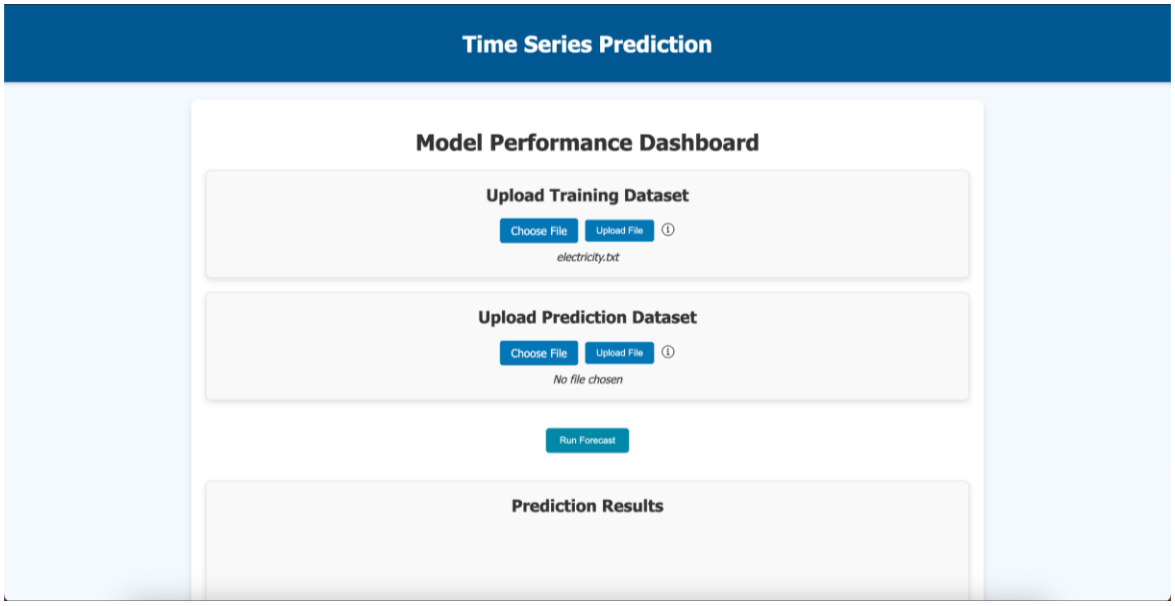
- The initial dashboard view of the Time Series Prediction application.
- The "Upload Training Dataset" and "Upload Prediction Dataset" sections are visible.
- Both sections have "Choose File" and "Upload File" buttons, along with an information icon for additional details.

Screenshot 2:
Tester name: Muhammad Abdullah Akif
Requirements covered: FR1



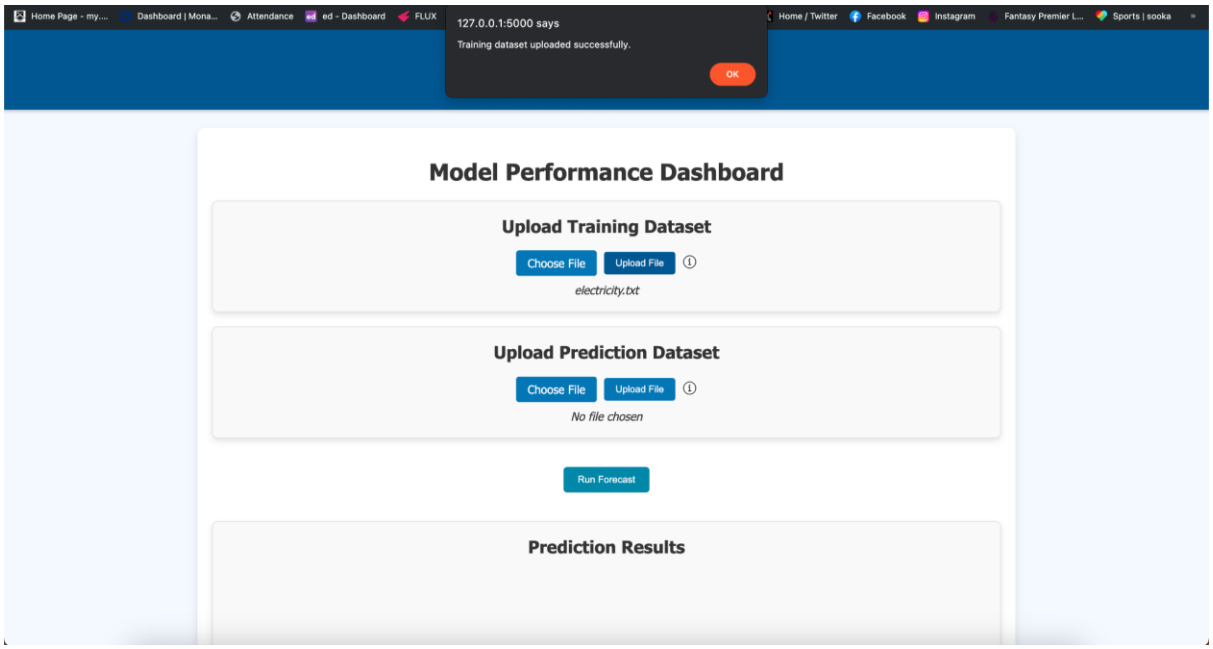
- The file selection dialog for choosing the training dataset.
- The user is selecting the electricity.txt file from the Downloads directory.
- The file content preview shows numerical data separated by commas.

Screenshot 3:
Tester name: Muhammad Abdullah Akif
Requirements covered: FR1



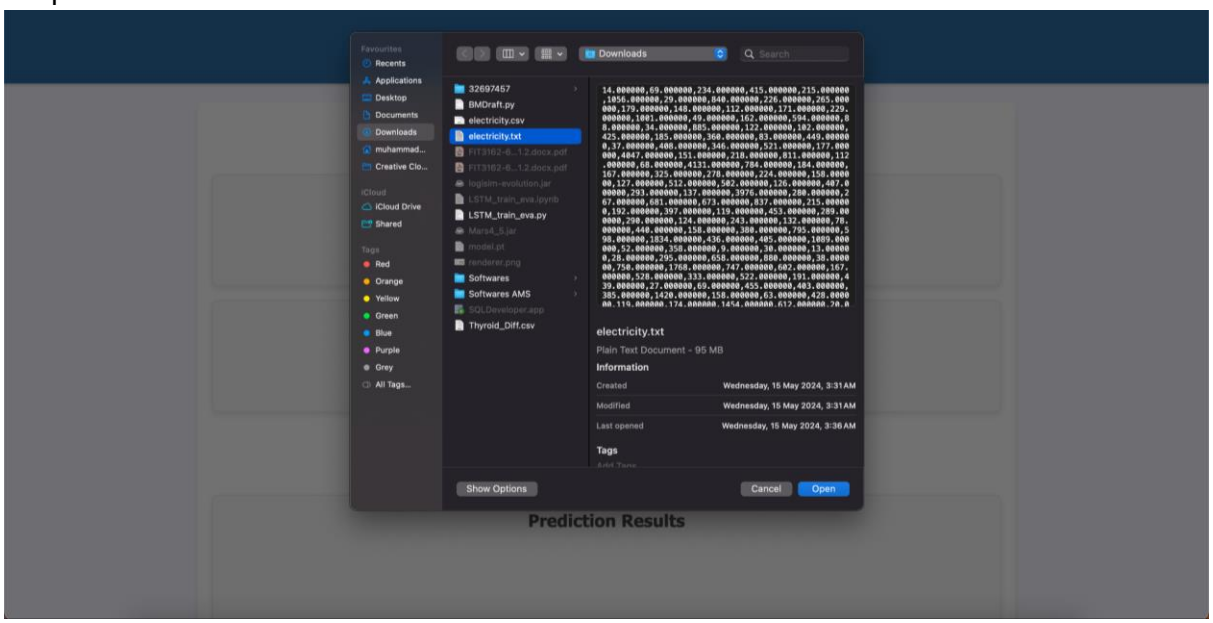
- The dashboard after selecting the training dataset.
- The "electricity.txt" file name is displayed below the "Upload Training Dataset" section.

Screenshot 4:
Tester name: Muhammad Abdullah Akif
Requirements covered: FR1



- The dashboard after selecting the training dataset.
- The "electricity.txt" file name is displayed below the "Upload Training Dataset" section.
- The message "Training dataset uploaded successfully" is shown in a pop-up.

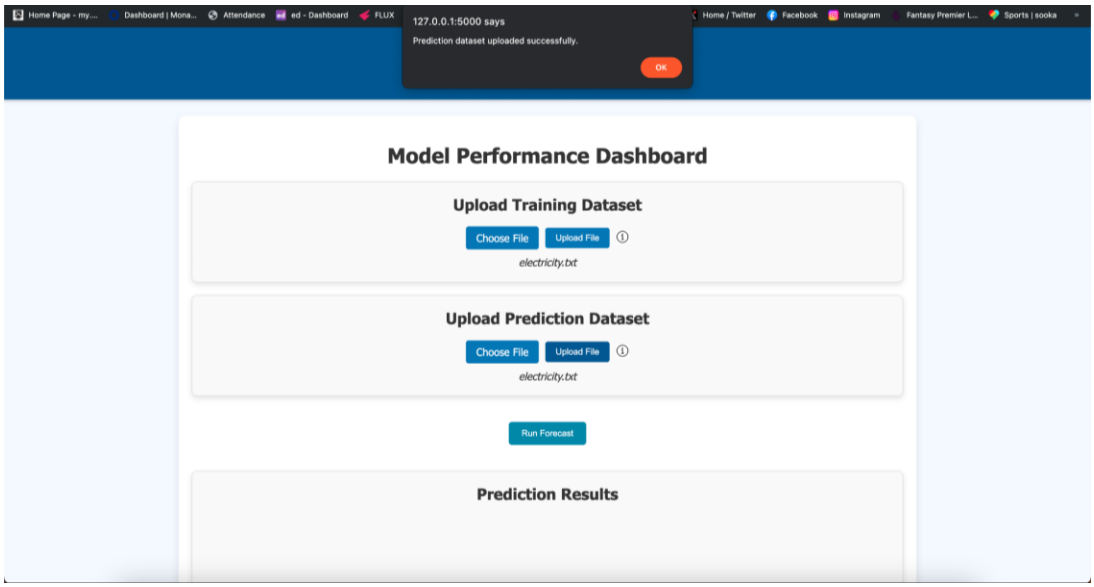
Screenshot 5:
Tester name: Muhammad Abdullah Akif
Requirements covered: FR1



- The file selection dialog for choosing the prediction dataset.
- The user is selecting the same electricity.txt file from the Downloads directory.

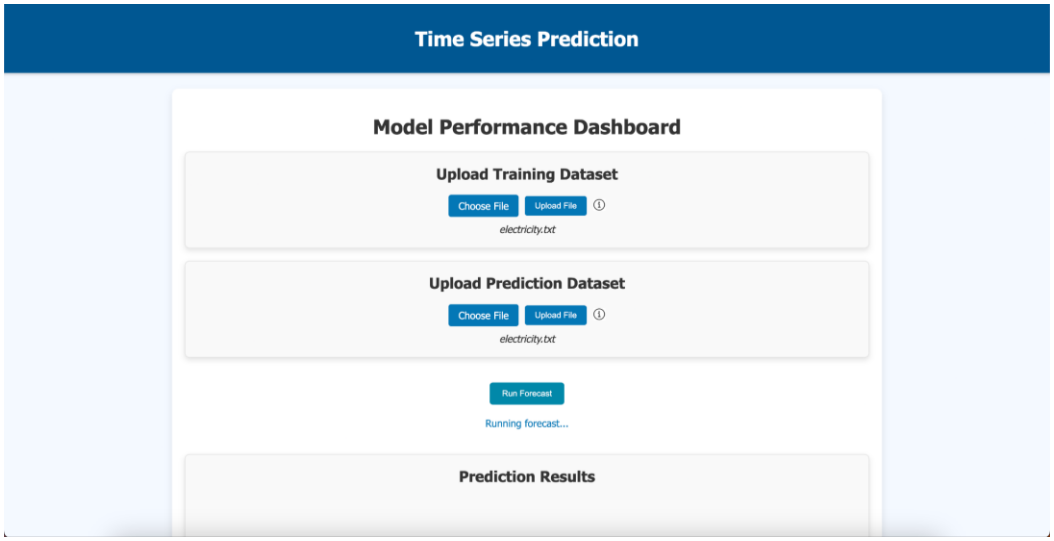
- The file content preview shows numerical data separated by commas.

Screenshot 6:
Tester name: Muhammad Abdullah Akif
Requirements covered: FR1



- The dashboard after selecting the prediction dataset.
- The "electricity.txt" file name is displayed below the "Upload Prediction Dataset" section.
- The message "Prediction dataset uploaded successfully" is shown in a pop-up.

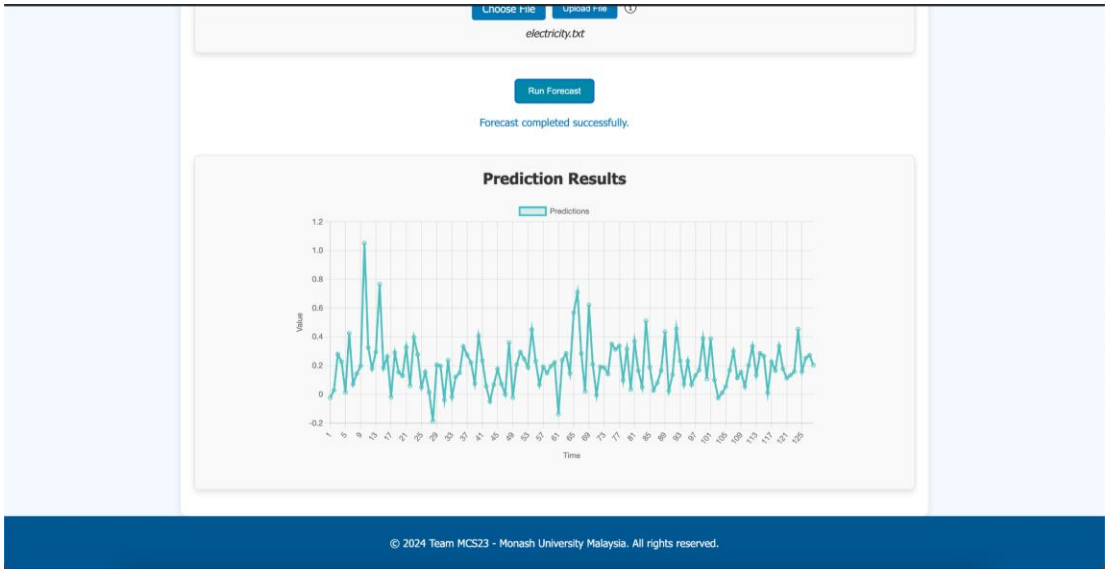
Screenshot 7:
Tester name: Muhammad Abdullah Akif
Requirements covered: FR3, FR4



- The dashboard shows the "Run Forecast" button after uploading both datasets.
- The message "Running forecast..." is displayed below the button indicating the forecast process has started.

Screenshot 8:
Tester name: Muhammad Abdullah Akif

Requirements covered: FR5

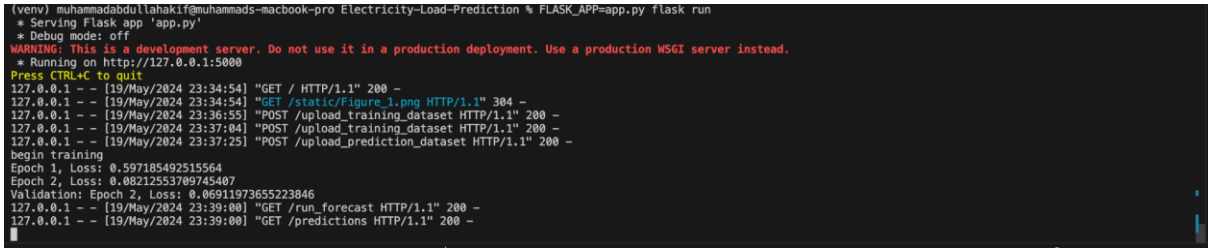


- The dashboard showing the prediction results.
- A line chart visualises the predicted values over time.
- The message "Forecast completed successfully" is displayed above the chart.

Screenshot 9:

Tester name: Muhammad Abdullah Akif

Requirements covered: NFR2



- Server Start:
  - The Flask application server is started with the command flask run, and it is running on http://127.0.0.1:5000.
- HTTP Requests:
  - Various HTTP requests are made to the server:
    - A GET request to fetch a static image (Figure_1.png).
    - POST requests to upload the training dataset (upload_training_dataset) and the prediction dataset (upload_prediction_dataset).
    - A GET request to start the forecast process (run_forecast).
    - A GET request to fetch the prediction results (predictions).
- Training Output:
  - The training process begins, and the loss values for each epoch are displayed:
  - Epoch 1: Loss = 0.597185492515564
  - Epoch 2: Loss = 0.08212553709745047
  - The validation loss after Epoch 2 is also shown: Loss = 0.0691367655223816.
- Completion:

- The successful completion of the forecast process is indicated by the HTTP status codes (200 OK) for the run_forecast and predictions endpoints.

Overall, this screenshot confirms that the server is running correctly, the datasets are uploaded successfully, and the model training and prediction processes are functioning as expected.

5. Useability Testing

To test our prototype's useability, we invited 2 participants who represent the target user to perform a series of tasks on using the prototype. The participants were first briefly introduced to what the test is for and what the prototype does, then they will perform the given tasks. The team will observe the participants as they test the prototype and note down their interactions. Finally, the participants will give feedback on their thoughts and areas to improve. This test covers the product requirement NFR5 to ensure that the prototype is user friendly, has a simple design and is easy to navigate.

Useability Test 1

Test description: Test if users can understand the guide on the type of data that can be uploaded.
Steps:

1. Read and understand the guide.

Expected output: User can read and understand the guide
Actual output:

Steps	Participant 1	Participant 2
1	Complete with no issues	Complete with no issues

Useability Test 2

Test description: Tests if users can fully perform a prediction using the prototype on his/her own.
Steps:

1. Choose and upload the training dataset.
2. Choose and upload the dataset for prediction.
3. Start prediction.

Expected output: User manages to follow the given steps to perform prediction using the prototype
Actual output:

Steps	Participant 1	Participant 2
1	Complete with no issues	Complete with no issues
2	Complete with no issues	Complete with no issues
3	Complete with no issues	Complete with no issues

Useability Test 3

Test description: Tests if users can understand the prediction results.
Steps:

1. Read and explain the graph.

Expected output: User manages to explain what the table and graph represents.
Actual output:

Steps	Participant 1	Participant 2
1	Complete with no issues	Complete with no issues

6. Test Limitations

Despite the thorough testing, the software testing is limited by few factors.

Time constraints
It is impossible to test every possible scenario, therefore our test mainly covers the main components of the prototype that allows the prototype to achieve its purpose of prediction. Besides that, due to tight deadlines, some product requirements are not included and not tested. For example, the requirements where the prototype accepts data files of various types and code is maintainable are not included and not tested due to time constraints. This results in some bugs or issues that users may encounter.

Test design
The test cases and designed and mostly carried out by the same team of people that developed the product. This leads to poorly designed test cases that are only focused on scenarios that the team has considered while coding and miss out some other scenarios. The team may also have bias towards their own code, assuming that their code is reliable. This leads to test cases that do not fully cover all edge cases.

Lack of testing skills
Most of the team did not take courses on software testing and lacked the specialised testing skills that professional testers possess. The team also may not be proficient at using the software tools. This leads to less effective testing strategies and limits the effectiveness of automated testing.

Environment dependency
Software can behave differently on different devices, different operating systems, different software and hardware configurations. The team can only perform in limited environments and may miss errors and issues that can occur in different untested setups.