Computer Science Capstone

CS 486 - Requirements Engineering

SmartState

Project Title: Event-driven MachinE Learning, Intelligent Assessor (EMELIA)

Final Report

Overview:

The purpose of this document is to summarize the development of our project and discuss the key functionality and architecture. This document will describe the components and modules that comprise our machine learning software.

Team Members:

David Rodriguez
Reed Hayashikawa
Andrew Hurst
Jianxuan Yao
Jesse Rodriguez

Capstone Mentor:
Fabio Marcos De Abreu Santos

Clients:

Jon Lewis
Aaron Childers

General Dynamics Mission Systems

Northern Arizona University
School of Informatics, Computing, and Cyber Systems

Version: 2.0
Date: 5.10.2020

1. Introduction

The development of this product was inspired by the U.S Coast Guard and General Dynamics partnership program, Rescue 21. Together these two parties have created an advanced command, control, and direction finding communication system. The motivation behind Rescue 21 was to use state-of-the-market technology to execute search and rescue (SAR) missions such as locating mariners in distress and saving live with a higher level of efficiency.

General Dynamics has implemented a large scale communication infrastructure throughout the continental United States to allow Rescue 21 to operate along all major waterways such as the Atlantic, Pacific, Great Lakes and other U.S territories. With this large infrastructure must come the ability to reliably maintain it. General Dynamics current communications system used by Rescue 21 relies on a ticket system to generate reports regarding system or hardware failures. These tickets contain data regarding:

- Code identifier for the error
- Length of time for which the error was present
- Location for where the error occurred
- How the error was received
- Description of the error
- Description of what was impacted by the system error
- Information regarding the resolution for reported error
- The component impacted by the system failure

Each error is classified according to the error type to assist system engineers in preventing future failures and restoring current ones.

The development of EMELIA was motivated by the current process of classification of tickets in their failure reporting, analysis, and corrective action system(FRACAS). In this system, System Engineers manually parse through the tickets and classify it in multiple categories.

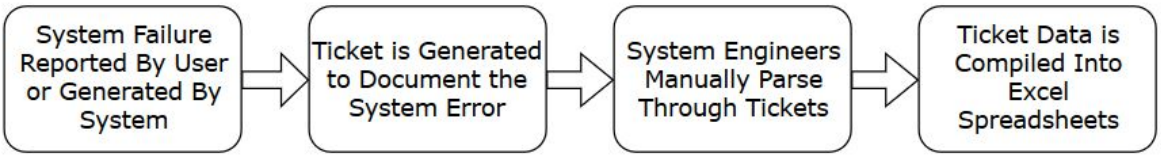

Figure 1: Overview of ticketing process

Specifically, the manual classification of a single ticket can be a time consuming process that requires two engineers to complete. A ticket takes approximately ten minutes to classify fully by a single engineer and must be agreed on by both; This presents the problem of added time when there are discrepancies between engineers.

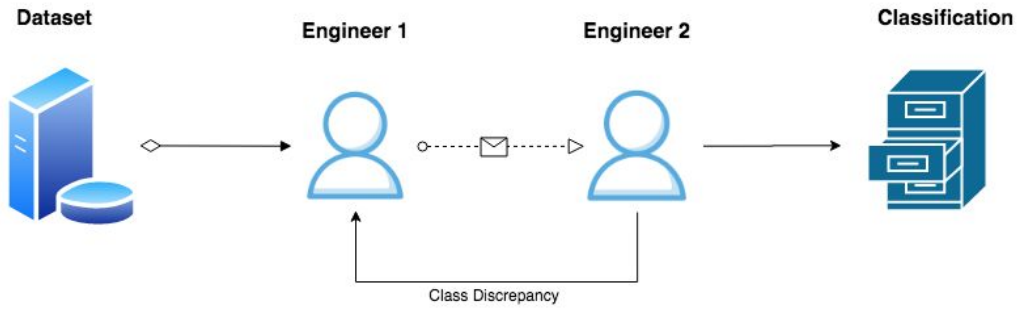

Figure 2: Classification process for engineers

Our envisioned solution is an Event-driven MachinE Learning Intelligence Accessor (EMELIA). EMELIA will automate the categorization of failure reports with machine learning in order to minimize time and resources significantly while still classifying at a high rate of accuracy. These tickets are rarely unique and most have common characteristics engineers use to categorize. This makes the data ideal to be trained and predicted with a Neural Network model. Similarly, this model reduces classification time from twenty minutes at best to fractions of a second at worse per ticket.

This report will outline the development process of this software, our requirements, the architecture and implementation, the testing of the system, and future work as-built.

2. Process Overview

The development process for this project included the use of task management for developers, work breakdown tools to determine work to be done, and version control to manage source code. To begin discussion of the development process, we will discuss the roles of each member followed by how the tools played a role in the development team workflow.

2.1 Team Member Roles

- David Rodriguez: Team Co-Lead, Technical Lead, and Release Manager. Primary responsibility pertained to task completion for project deliverables, core development of EMELIA functionality, and management of all changes to source code in the GitHub responsibility.
- Andrew Hurst: Team Co-Lead. Primary responsibility included work on word documents and assisting with architectural overview of EMELIA.
- Reed Hayashikawa: Team Architect, Recorder, Release Manager. Primary responsibility was to assist in development and testing of EMELIA functionality. Reed also assisted in reviewing all changes to source code.
- Jesse Rodriguez: Team Unit Tester. Jesse was charged with overseeing development of tests for all functions in the data processing module.
- Jianxuan Yao: Assisted with unit tests and word documents.

2.2 Development Tools

The team utilized tools such as Bitrix21 to manage and delegate tasks. This tool proved to be difficult due to the user interface, but provided the opportunity to utilize a tool for development at no cost to the team.

The most important tools for project development are Git and GitHub. These tools proved useful for managing new features and changes to source code. GitHub allowed participating members of the team to review and provide feedback for all code development. Version control proved to be a challenge for most of the team due to inexperience. GitHub and Git streamlined the work process for making changes to the codebase, regardless of the initial learning curve required by the team.

For more information regarding the project please visit: https://github.com/David-Rod/EMELIA

3. Requirements

This section will outline the requirements that were instructed in the requirements document and focus on the standards used in order to construct EMELIA. The requirements section will be organized in the following subsections:

1. Functional Requirements
2. Non-functional Requirements

The requirements have been obtained through weekly meetings. The functional requirements will outline features that are present within EMELIA.

3.1 Functional Requirements

1) EMELIA shall train test data and be able to classify tickets
    a) The system shall be able to use test data to train the classifier
    b) The training model shall classify ticket data using categories from tickets that have already been classified
2) EMELIA shall be able to receive and process files containing ticket information
    a) The system will analyze information in the CSV file and format the data as input to be read in by EMELIA
        i) Organizing data is essential for specifying data columns used for classified output data
    b) The system will utilize a supervised learning model
        i) The system will use domain data to create a neural network model for the purpose of machine learning
        ii) Provided data is defined prior to classification, and will be used to produce a predefined output
            (1) Ticket data is labeled and the total number of classification types are determined prior to the learning
        iii) The system will use the TensorFlow framework as a training model
            (1) A neural network has been the proposed machine learning technique for this project.
            (2) TensorFlow will allow for the neural network to utilize backpropagation to adjust the input as it is classified.

3) The system will test the model's accuracy and performance

    a) The system shall test the model's accuracy during training

    b) EMELIA needs to classify ticket data at a rate of 90% accuracy

    c) EMELIA need to reduce time spent on each ticket

        i) A total of 20 minutes is the average time taken to classify ticket correctly

        ii) A total of 30 minutes is the average time taken to classify and correct a misclassified ticket

    d) The system shall be able to benchmark the system's performance for training of a particular data set

        i) Returns the benchmark data to the user through the command-line interface

            (1) Number of items classified

            (2) Filename of the data that was classified

3.2 Non-Functional Requirements

This section will cover non-functional requirements needed for EMELIA. These non-functional requirements have been taken from the functional requirements stated in the previous section. Each of the non-functional requirements will be detailed according to priority.

1) Accurate

    a) The system shall be able to classify data at 90% rate of accuracy

        i) This system needs to be able to classify system failures accurately using the ticket data

(1) Accuracy and precision needed for resulting data

ii) This non-functional requirement is the most important for this system due to the risk associated with inaccurate results

2) Reliable

a) This system shall be accurate and dependable

i) The system shall not only produce accurate results, but shall produce the same relative result on the same data set

(1) This will be part of the testing process for the system to ensure its viability

3) Retainable

a) System shall be maintainable and usable for a long term.

i) This solution shall be the start of a larger classifier for our clients communications systems

ii) Retainability will be dependent on the quality of the architecture, accuracy, and reliability of this project

b) The system shall be able to integrate successfully with the existing framework

i) The team will be utilizing CSV files to extract and classify data

ii) The team may need to refactor the working version of the assessor to tie into the existing software

(1) Client currently uses software which contains individual ticket data

4) Extensible

a) The system shall allow for changes to the classification process by adding and removing parameters.

    i) These changes parameters shall allow the engineers to increase the classification accuracy for the training model

5) Scalable

    a) The engineers that use the system shall also be able to change the number of items that are classified within the data set

        i) If an engineer would like to classify a subset of the data rather than the entire data set, the system shall be flexible enough to allow:

            (1) Time efficiency by reducing the total time spent on classification of ticket data

            (2) Serve the business needs of our client by accommodating multiple ticket systems.

                (a) Usable outside of the current ticket system domain

6) Testable

    a) The system shall have functionality that allows for the data classification process to be tested

        i) System engineers that use the classifier shall be able to run a subset of their data through the training model to attain a result

            (1) Pass the same subset to the test suite to ensure that they are receiving accurate results

        ii) Testing shall determine accuracy and precision of data classification

Overall, EMELIA seems to have the functionality to satisfy all functional requirements. All features agreed upon between the development team and the client have been satisfied. This product facilitates acceptance of ticket data files and processes the files so that data can be passed to the learning model. The learning model is able to effectively train on the data and functionality is in place to ensure that the learning model has been trained correctly. Most importantly, the model can accept alarm data with corresponding tickets, and effectively classify those tickets. The nonfunctional requirements have been met for this project. Although the functionality provides a foundation of actions to be taken, the system is organized in a way to expand the codebase and add to specific modules. The three most important nonfunctional requirements have been met, which were to ensure system accuracy, reliability, and extensibility. This product can become even more refined and serve further needs of the client if more development iterations were to be given to this product.

4. Architecture and Implementation

4.1 Overview

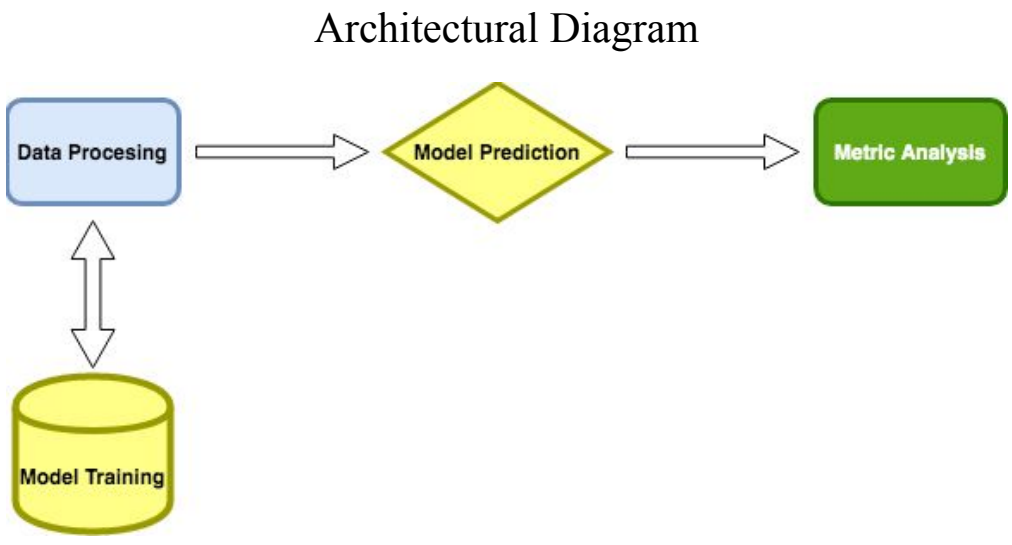

Figure 3: Overview of EMELIA

4.2 Key Module Features

Understanding EMELIA requires a basic understanding of the key responsibilities and features of each module. The system is composed of four major components: Data Processing, Model Training, Prediction Model, and Prediction Metrics. The combination of all these components produces an accurate predicting system for General Dynamics Error Reporting Ticketing system. Each module plays an important role in that process.

The data processing module essentially formats the chosen tickets and the relative information in such a way for the Neural Network Prediction Model to compute. However, the data formatting will vary slightly for input for the prediction model and training the model. The Prediction Model will take the formatted data and use machine learning to predict the corresponding classifications for the data. The Metrics Analysis Model will take the output from the Neural Network and consolidate it in a readable fashion, producing output files and provides metrics on the system's performance. The last component, the Training Module, will not be essential to the process every time the program runs. It will only be used for initially training the Neural Network and retraining when new features to predicting are introduced.

4.3 Control Flow and Communication

Control of the program will be dictated from a command-line environment. Arguments will be passed from the terminal to the main driver program. The EMELIA architecture will be structured in a procedural fashion; Each module's functionality will be called from the driver and data will flow through there between modules.

4.4 Architectural Influences

The architectural influences shaping the software are a high percentage of procedural architecture, as previously mentioned. This is a result of our programming requiring more of a

pipeline of processing structure. However, the Neural Network model will be organized as an object allowing the user to store, save, and recreate. Overall the system will implement procedural methods with small object-oriented influences.

4.5 Module and Interface Descriptions

This section will provide detailed descriptions of the modules and interfaces that will be utilized in creating EMELIA. The interface of this system will only include exporting of data and/or functionality and importing of data and/or functionality depending on the responsibility of the module. There will be no graphical interface for this system. All commands will be ran via a command line. The purpose of this system is to allow each module to access data or functions in a sequential manner that will make troubleshooting easier for development.

Included with every module is a detailed description of the correlation that the module has with the other components of the project, interface mockups, and overview of the coding practices needed to have a successful implementation. Before discussing each module, we have developed the following system diagram for the entire process that our project will undergo from data processing to output of the data.

See Figure Next Page

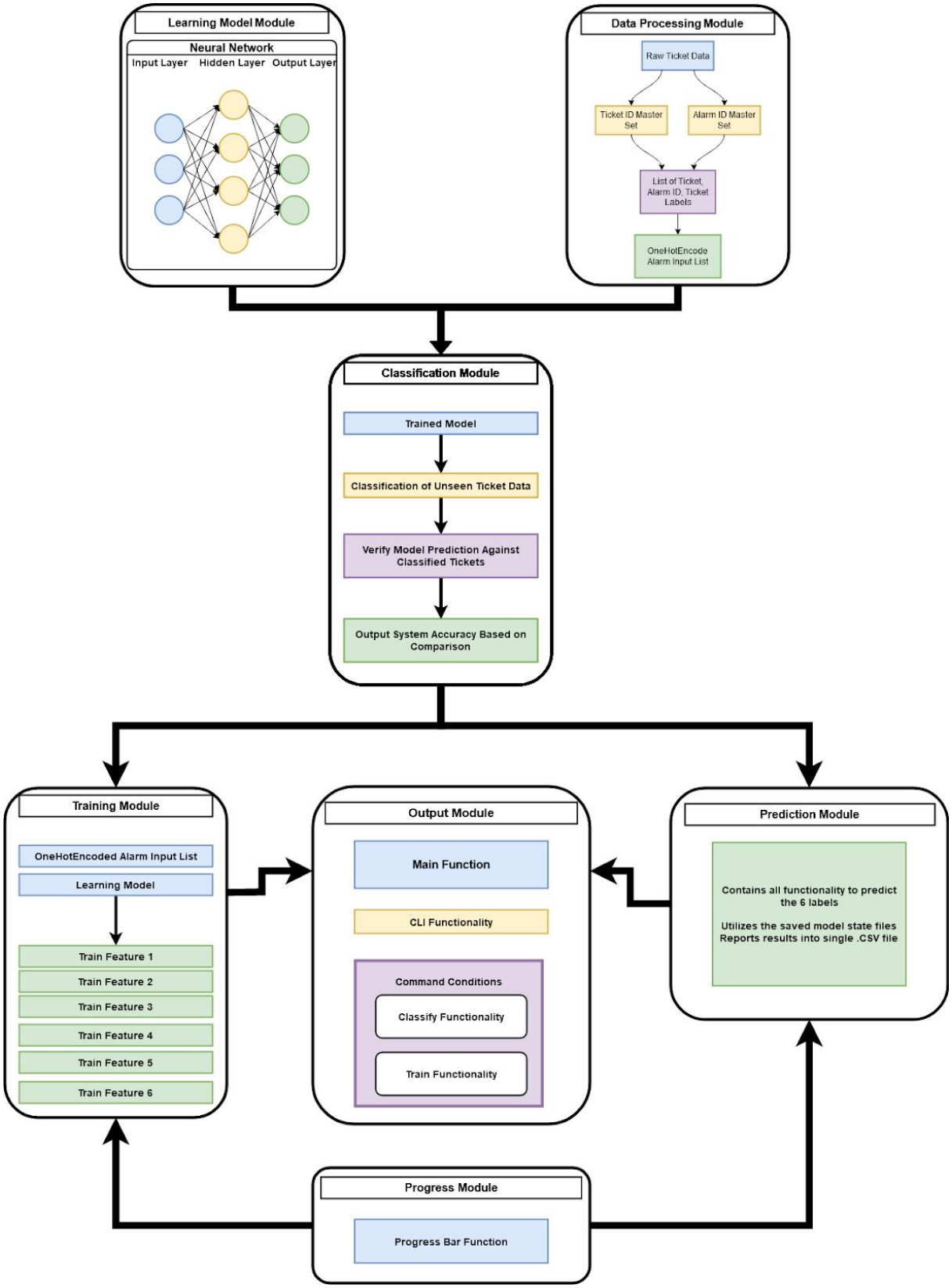

Figure 4: System diagram for EMELIA

4.5.1 Data Processing

Component Responsibilities

This component will be responsible for acquiring all necessary ticket data and alarm data needed to pass to the neural network layers located in a different module of the system. This module will configure the data properly using basic coding practices in order for it to be passed to the neural network. Our system will need to be able to receive input data in the format of a CSV file representing several “tickets”. We will then retrieve this data by implementing a program to read in the arguments through the command-line interface. Below is a diagram of the two csv files and the relationship that exists with the contents of the files.

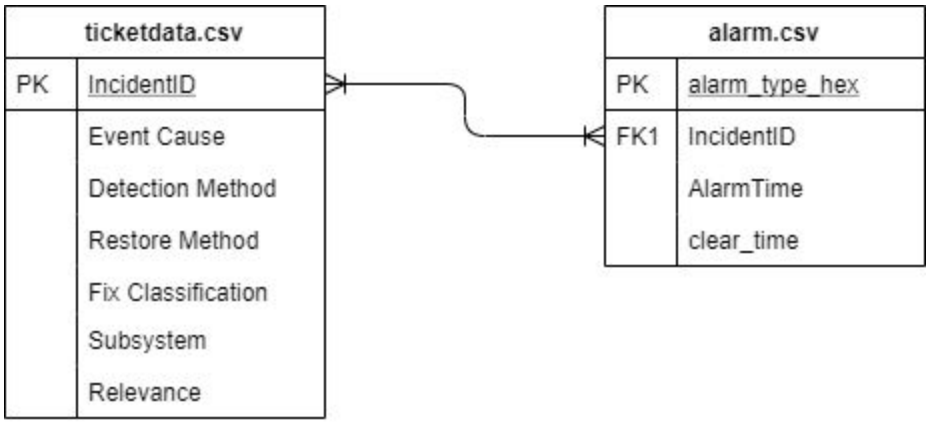

Figure 3: Relational diagram for ticket data and alarm data

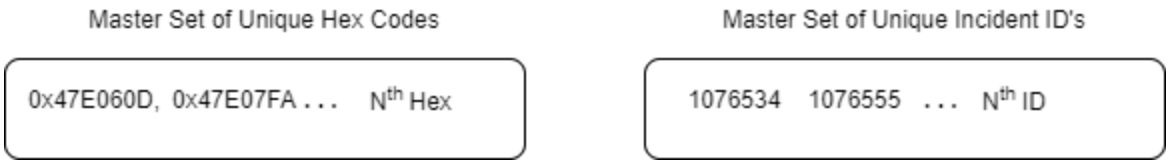

Figure 5: Unique hex and ID values stored in sets

Since the ticket will be in two csv files this module will have to parse through each “ticket” and join the two tables of data to make all incident identifiers for tickets and alarm hex codes available in a single file. These tickets will be similar in structure but may differentiate by the

number of classifications or category values as seen in Figure 3. With that being said, this module will need to create data structures containing both ticket and alarm data that can be used within this module along with the other modules. In order for this structure to be accessed in the future, this module will store OneHotEncoded ticket data in an array. The program will need to utilize function calls to OneHot encoded ticket data that will serve as input to the learning model.

The following figure is an example of the array structure that will be implemented to store corresponding ticket data:

```
[
        [ID1, [0x23456], [Label1, Label2, Label3, Label4, Label5, Label6]],
        [ID2, [0x23457], [Label1, Label2, Label3, Label4, Label5, Label6]]
        [ID3, [0x23458], [Label1, Label2, Label3, Label4, Label5, Label6]],
        [ID4, [0x23459], [Label1, Label2, Label3, Label4, Label5, Label6]],
        [ID5, [0x23450], [Label1, Label2, Label3, Label4, Label5, Label6]],
        [ID6, [0x23451], [Label1, Label2, Label3, Label4, Label5, Label6]],
        .
        .
        .
        [Nth ID, [Nth Hex Values], [Nth Label Values]]
]
```

Figure 6: Unique ID's with Corresponding Hex Codes & Labels

The purpose of this array will be to store all related data in an ordered fashion, while remaining accessible to developers. The values in this nested 2-Dimensional array will allow the developers to OneHot encode its contents, and make comparisons to prediction data in later modules.

Program Module Relationship

This module has a direct relationship with other components and advances data to downstream modules. During this processing component, a master list of data will be created that will be used

for relational mapping in other modules. In order for the training model, which is the next module in the system, to be accurate the data needs to be configured correctly and thoroughly examined to prevent loss of training data and reduce bias in the data provided to the neural network model.

Program Interface

This module will interface with the other modules in the system by moving data down the pipeline. Data will be stored in an array and exported to the next module for use. Since this module is the root of all functionality for the other modules, it is crucial that we implement this module so that it organizes data in a way that accurately reflects each ticket. This is to ensure the data can be used for the training module and be successfully displayed for the output metrics.

4.5.2 Learning Model

The neural network model will be used to train on the test data provided by the client. The learning model will be contained within a single function, inside a module, to be exported for use by each of the features that will need to be classified for the ticket system.

Component Responsibilities

Construction of the learning model is the primary responsibility of this component. The learning model will be constructed within a single function. Each of the models will be part of a larger pipeline that feeds data into a learning model.

The number of dense layers will not be dynamically created. Since the learning model will be "tuned" to create the highest accuracy results, the user(s) will not be able to control the number of dense layers or specify a different activation function for the model. The activation function will be a "softmax" function in order to generate confidence value prediction data as an output of the model.

Program Module Relationship

This module will be exported to the model classification module within the system. This module is only meant to construct the learning model. By modularizing the system in this way, the learning model can be adjusted by future users and engineers without negative side effects or need for reconstruction. The implementation for this module provides the highest amount of integration and the least amount of coupling to the ticket data and the resulting classification of ticket data.

Program Interface

The interface for this module will be easy to implement and use by the design team. Since Python is the programming language used in this project, the team will need to import the file containing the learning model and make a single function call to a function containing the learning model. This design will be most scalable for future development and allows engineers to import and run the learning model for classification or perform operations in the module without the learning model.

4.5.3 Classification

This model will have the data processing module functionality and functionality from the learning module imported to perform work on the output. This will include helper functions that format data and have functions to check the validity of the learning model when training on data and generating data using test data.

Component Responsibilities

This component contains the learning model that will be stored for each feature that will be classified regarding each ticket. The model will be trained and store the state for each of the neural network models which are specific for each feature. The state of the learning model will be saved to a .md5 file or .hd5 file. Each model will be wrapped inside a function and will be passed ticket feature data via a parameter. By bundling each of the trained neural networks in a function, these trained networks can be exported to the output module for comparison to the test data.

Program Module Relationship

The module will be responsible for importing the neural network function created in the previous module. The function that is imported will be used on six separate occasions to train on each feature of ticket data. The ticket data will be imported from the data processing module as an array. The array will contain OneHot encoded values that will be used as input for training. Data processing will be a prerequisite for this step to successfully train on the data that is provided. This module will be the third step in the pipeline of data. The file will construct and export its functionality to be used downstream. It simply exports all the functions used to train the neural network learning models to the next available module.

Program Interface

This program will export all functions to be used by the output module. The functions that contain the trained learning models will be explicitly named so that developers can differentiate between them in the driver file. The functions from this module are not codependent and may be used in any order within the driver file.

4.5.4 Training

All training on data will take place in this module. Functionality from the classification function will be used in this module to generate the state files of saved model state for each of the classifications that require accurate labeling.

Component Responsibilities

This component will take a subset of the provided training data and allow the learning model to train on the input and corresponding output. The subset of training data is randomly generated and trained upon with the constructed module. This module calls the classification function for each label and saves the state of the learning model in the .hdf5 file that corresponds to the label. Next, a set of predictions are made using the remainder of the training data not used for training. Finally, the validation function is called to determine the accuracy of the system by comparing the index of the correct label to the index of the confidence value generated by the learning model. If the positions match, the system correctly trained on that set of data.

Program Module Relationship

The functionality in this module will be called in the output module. This will be part of the control of execution provided in that module. Training is an expensive process. It requires several minutes of execution for the learning model to successfully train on all the data points provided on the training data set.

Program Interface

All actions in this module are wrapped in a single function that accepts various parameters provided in the output module. The 'training' action must be specified in the output module for this component to execute.

4.5.5 Prediction

This component contains all functionality to generate predictions based on a single input file. The file is fed into the module, where function calls extract specific fields as input in an attempt to generate label predictions for each classification column pertaining to a ticket.

Component Responsibilities

This component will generate an output CSV file that contains ticket 'Incident ID' and corresponding classification labels. This resulting file contains the 'Incident ID' field and the 6 other label columns needed for each ticket. The name of the resulting CSV is specified by the user as a parameter in the output module. This module is only executed if the user of the program specifies it as an action.

Program Module Relationship

The prediction module is dependent on the parameters passed to the output module. If the action is specified in the output module, the function for this module will be executed to generate the expected results. Compared to execution time of the training module, this module is significantly cheaper to execute. This is due to the fact that the model will be executed frequently on a small subset of tickets.

Program Interface

This module interfaces with both the progress module and the output module. The functionality from progress is imported to this module. The functionality from this module is exported to the output module.

4.5.6 Progress

This module contains a single function to display execution progress to the user.

Component Responsibilities

Render a progress bar to the user while the program executes. This ensures that the user will not mistake the runtime needed for the learning model to train is not the result of a bug.

Program Module Relationship

The functionality in the is module is exported to the prediction and training modules.

Program Interface

In both the prediction and training module, a single thread is created from the main process of execution. This single thread is used to execute the progress bar function, while the core action of the program is executed. The thread is required so that both functions are executed concurrently.

4.5.7 Output Metrics

Component Responsibilities

This module is responsible for accepting user commands via the command line interface. This module has the ability to process a basic set of commands that control the flow of execution.

Program Module Relationship

This module will import functionality from the prediction and training modules. As the user passes parameters via the command line, this function will then relay those parameters to the prediction module, training module, or both depending on the action specified by the user.

Program Interface

This module will utilize the functionality from the other modules in the system, and the command line, to determine which process to execute within the program. The module will accept a single flag command and up to four file names to complete any action capable of the system. The files provided can include:

- Input alarm file
- Input ticket file
- Test ticket file
- Name of file that will contain results for the system

To understand more about the input for this module, please refer to the 'Help' section of the 'Product_Delivery_User_Manual' document.

5. Testing

This section will discuss our team's testing plan where we tested our software to ensure that it functions properly. This section will also explain our team's process for ensuring that important functions are performing as we expect them to in order to ensure the quality and robustness of our code. Lastly, this section will be organized into each testing strategy then the overall results of our tests.

5.1 Testing Strategy

This section will focus on our three main types of testing: unit, integration, and usability. It will also cover the results of our test cases and what overall changes we made to the software.

5.2 Unit Testing

Unit testing focuses on the tests that have the most influence on the overall behavior of our system, checking that the client's data is processed correctly before it is utilized by the training model ensures that the results of the training model are useful to our client.

Since the framework of EMELIA uses python, the testing tool that our team utilized was the python library Unittest that allowed us to test the individual components of our software. In order to determine the overall effectiveness of each test, we also utilized coverage.py which provides metrics on the effectiveness of each test case. The main components of EMELIA that were tested involve the processing of data into a usable data structure, the values that we are encoding, and the one-hot encoding functionality that prepares the data to be passed into the learning model for training. Unit Testing was important for the development of EMELIA as it allowed the team to prove that our software is effective in solving the clients' problem.

5.3 Integration Testing

While unit testing focuses on individual components and validates proper functionality of those units, integration testing is primarily responsible for testing the functionality and interaction between major components. As a result of our pipeline framework, changing one component could significantly affect the way that further components interact and cause errors for multiple components. Our system consists of four main modules: Data Processing, Training Model, Model Classification, and Output Metrics. The following points depict how data is sent between modules:

- Data Processing → Training Model
- Data Processing → Prediction Model
- Training Model → Prediction Model
- Prediction Model → Output Metrics

Ensuring that the correct transmission of data occurs from one module to the next is necessary to show that our system is reliable. Integration testing is important to our software because it allows us to catch errors in one module that could adversely affect the performance of the other modules.

5.3 Usability Testing

Usability Testing is primarily focused on the overall user interaction with the product. The user's ability to interact with our product will determine if our system is useful to the client. The main focus for usability testing in our software is the accuracy of our ticket classification, how easy it is for users to perform actions using our system, and the speed that the user completes their desired task. EMELIA is a system that is designed to reduce the amount of time spent in our client's current ticketing system classification process. If the user cannot accomplish each use case within a relatively short amount of time, the usability test for that specific use case is considered a failure and our team must make improvements to our software. Additionally, since our client's system is proprietary, an expert review from our client will provide us insight regarding the accuracy of EMELIA and suggestions regarding the framework and functionality of the learning model. The use of Expert Review is an important part of our usability tests because of the impact it will have on the functional requirements of this project. In order to test the non-functional requirements of EMELIA, additional review by non-experts could be useful in determining how a user interacts with the system.

5.4 Results

The Unit test cases resulted in minor refactoring of the Data Processing module that included the handling of NULL values for ticket labels. The results of integration testing resulted in a major overhaul to how the Data Processing module works and interacts with the remaining modules. The results from our usability test cases is the refactoring of the command line interface to include additional information to the user. Overall, these test cases assisted in the development of our software because it allowed us to detect and catch problems early so that they could be dealt with in a timely manner.

6. Project Timeline

This section will detail the timeline of events during the development of EMELIA and include the key phases. The following Gantt chart details the overall project timeline:

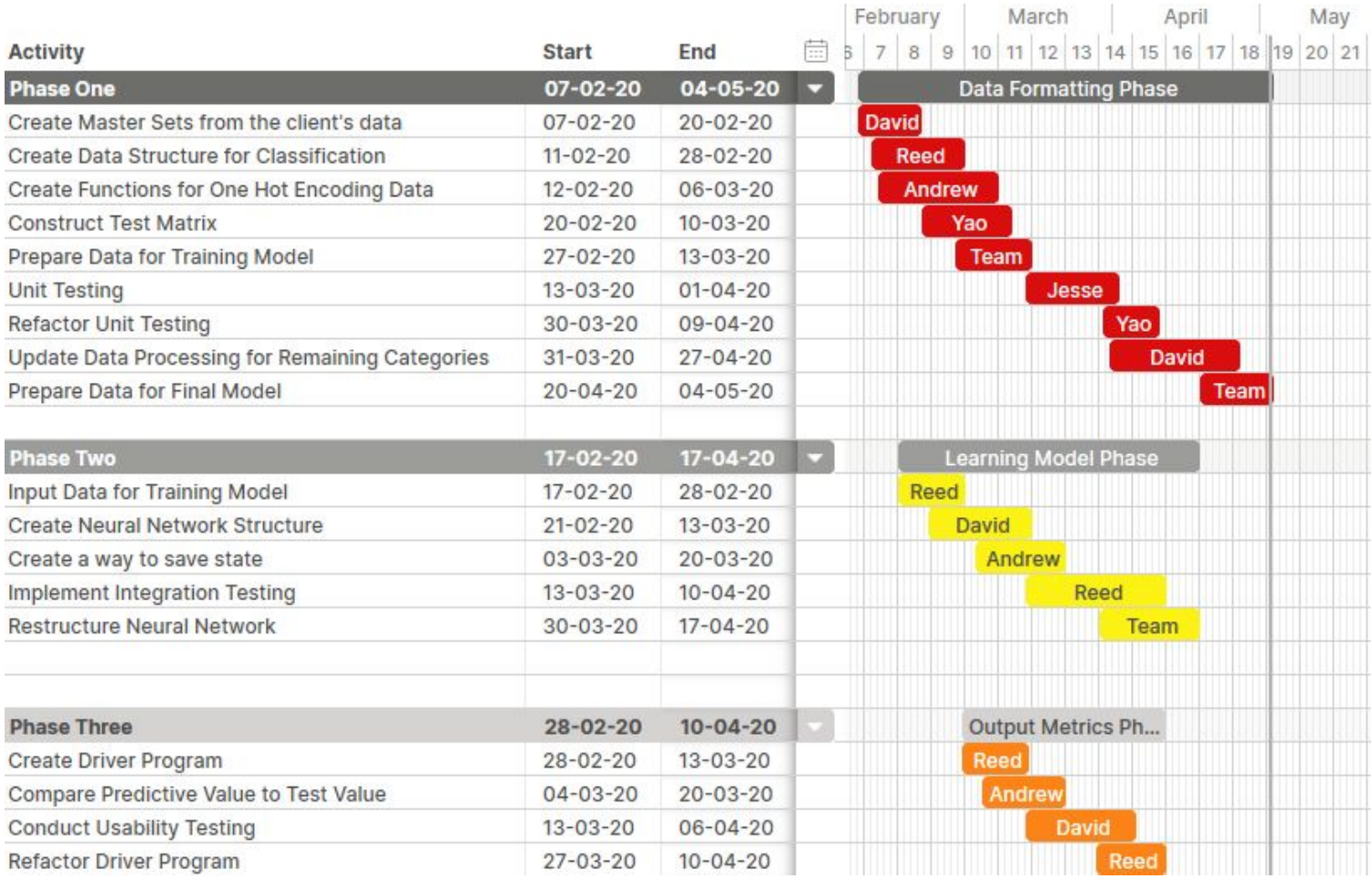

Figure 7 : Gantt Chart

6.1 Phase One

This phase is focused on the Data Formatting Module where the client's data is formatted into a dataset that we can use to train the model. The Gantt chart details the tasks and who was in charge of its completion. Our team lead, David, was responsible for assigning tasks to individual team members and the creation of the one-hot encoding functions for converting the data into its binary representation. In this phase, David was responsible for creating the master sets from our client's data and updating data processing to include all categories of classification. Reed and David were responsible for the creation of the data structure and assisting with the preparation of the model for demonstrations. Jesse was responsible for conducting and consolidating the Unit tests developed to test the functionality of individual components. Yao was responsible for refactoring the existing unit tests after they have been consolidated. Our Team is currently finishing up the last task of Phase One as we prepare our model for final delivery to our client.

6.2 Phase Two

This phase is focused on the Training Model and Model Classification modules where the ticket data is passed into the trained model for classification. Reed conducted research and implemented the functions needed to save the state of the learning model after it successfully trains. David was responsible for creating the neural network structure and assisting in refactoring of the model to increase accuracy. David was responsible for inputting the prepared data into the training model and implementing integration testing for all modules.

6.3 Phase Three

This phase is focused on the Prediction and Output Metrics modules where the results from the learning model are compiled and the accuracy is determined by comparing the predictive values to the test values. Reed was responsible for comparing the predictive value to the test values that were obtained by the learning model. David was in charge of conducting usability tests with our

client to ascertain the software's ease of use. David was also responsible for refactoring the driver program to run the main functionality of each of the modules.

7. Future Work

This section will focus on the future development of our client's software by introducing new features that could be implemented by future Capstone teams.

7.1 Command Line Interface

EMELIA may require a more robust Command Line Interface (CLI) to perform more actions in the future. The implementation could be changed to use a more sophisticated and customizable CLI, such as the Python Click library.

7.2 Increased Commands

EMELIA currently accepts four commands to perform user actions, one of which is the help text for the system. The functionality may expand and utilize more parameters in order to complete more tasks or accommodate new features.

7.3 Accept More Ticket Features

The system could increase the accuracy by accepting more ticket features. Currently, EMELIA uses functions that target alarm hex codes to predict ticket classification labels. A large improvement for increased accuracy would be to use other ticket features such as site type as an input to improve learning model performance.

8. Conclusion

This document is a detailed view of the motivations, development process, and the as-built product. The product as-is represents a strong-beta on the target software. The sections above provide a comprehensive explanation on how the product is configured, the working components, and essentially a resource to get any future parties up to speed. This will give the sponsors a strong basis for future refinement to a polished final product, or more likely architectural/programmatic strategies to implement in their current architecture.

As General Dynamics has stated, the categorization process of error reports has a clear potential of streamlining efficiency. EMELIA has the capability to save a significant amount of time, improve accuracy, and increase throughput of classified error reports which provides system engineers the ability to maintain the communication system of Rescue 21 at a very high-level of reliability. Foreseeable impacts of this system also include the potential to be implemented for other kinds of error reporting systems in general.

SmartState as a team has learned much about the development process and an extraordinary amount about the Software Engineer industry in general from this Capstone program. We consider this knowledge and experience to be invaluable and would like to thank General Dynamics for providing wonderful sponsors as peers and mentors during this process. Our team has worked incredibly hard on this project and are proud to walk away as better engineers. We hope that our product will remain reliable and maintainable, now and in the future, for our client.

Appendix A

A.1 Hardware

To start development of EMELIA a user should have hardware that meets the minimum requirements to run the TensorFlow machine learning library. The operating system for a user machine should be Windows 10, since all end users will be using Windows 10. Linux and Mac OS are acceptable for development, but all usability testing should be performed on Windows 10. The central processing unit (CPU) should be a minimum of generation i3 (Intel) or other brand equivalent with 2 cores, 4 threads, and Advanced Vector Extensible (AVX) compatible. Main memory for a machine should be 4GB or more to avoid prolonged runtime.

A.2 Toolchain

The software development team used a variety of tools throughout the development cycle. The following editor/IDE's were used to develop EMELIA:

- Visual Studio Code (VS Code)
- Jupyter Notebooks
- Spyder
- Notepad++

Visual code has an advantage over the other editors due to available plugins. All development was developed within an Anaconda development environment. Anaconda (or Conda) is a package management system that manages dependencies that are required to develop EMELIA with TensorFlow. The following are the basic packages and version numbers needed for development:

- Python 3.6

- TensorFlow 1.13.1
- Keras 2.2.4
- Pyodbc 4.0.26
- Pandas 0.24.2
- Numpy 1.16.5
- Ipython 7.4.0
- Scikit-learn 0.20.3
- Tqdm 4.39.0

Python is the required programming language. TensorFlow is the required machine learning framework and Keras is a TensorFlow library with built-in optimization tools. Pyodbc is an open database connectivity tool used by the client to process ticket data. It is not used in development at the moment, but should be included in the package management system. Pandas and Numpy are data analysis tools that support multidimensional matrices needed for manipulating ticket data and providing input to the machine learning model. Ipython is an interactive python shell that allows for testing code inside the Anaconda Command Prompt. Scikit-Learn is a machine learning library for the Python programming language that contains built-in functions for generating training and test datasets. Tqdm is the progress bar library which provides a progress indicator as the learning model moves through different stages of execution.

A.3 Setup

Setup to begin EMELIA development can be accomplished in a few steps. The first step is to install the Anaconda package management system. This system provides a variety of tools that can be installed from the Anaconda Navigator application. Tools such as VS Code, Jupyter Notebooks, and the Anaconda Command Prompt can be installed directly from the Anaconda Navigator portal. Developers must create a new development environment prior to installing packages. As mentioned previously, all packages listed in the Toolchain section of this document can be installed using the Anaconda Command Prompt. Next, developers must clone the repository from GitHub. Establishing a GitHub account and an SSH encrypted key are also

required, but those instructions are beyond the scope of this document. Once a development environment has been activated, all packages have been installed, and source code has been cloned to the developer's machine, the developer may launch their selected editor and begin development.

A.4 Production Cycle

Production for EMELIA is straightforward. This project was meant to determine if ticket classification is viable as an end product, without access to the client database or using real ticket data. All data provided to the development team has been "cleaned" of any proprietary information, so EMELIA is more of a proof of concept for General Dynamics. Due to these circumstances, the product can be executed directly from the command line to begin training, testing, and ticket classification. The file responsible for providing ticket classification results is output.py. The project can be ran using the following commands:

```
python output.py --Help

python output.py --Classify <CSV Alarm Filename> <CSV Ticket Filename> <CSV Test Alarm Filename> <CSV Prediction Filename>

python output.py --Train <CSV Alarm Filename> <CSV Ticket Filename>

python output.py --Both <CSV Alarm Filename> <CSV Ticket Filename> <CSV Test Alarm Filename> <CSV Prediction Filename>
```

These commands allow the program to perform its intended actions using a single flag command ("--Flag") and a maximum of four files. The "--Help" flag provides help text to get started with running the program. A detailed view of all commands can be seen with this command. An example:

```
================== START HELP TEXT ==================
Please choose the operation needed to be performed by the system.
Commands should be followed by the files to be used.


Commands to be passed:


TO CLASSIFY ONLY:
python <Filename> --Classify <CSV Alarm Filename> <CSV Ticket Filename> <CSV Test Alarm Filename> <CSV Prediction Filename>


TO TRAIN ONLY:
python <Filename> --Train <CSV Alarm Filename> <CSV Ticket Filename>


TO CLASSIFY AND TRAIN:
python <Filename> --Both <CSV Alarm Filename> <CSV Ticket Filename> <CSV Test Alarm Filename> <CSV Prediction Filename>


=================== END HELP TEXT ===================
```

The other two flags are named to represent the action to be taken. “--Train” will process ticket and alarm data to update the saved model state. “--Classify” will accept an alarm hex code file and produce prediction results using that file.

Any edits made to the source code are dependent on the developer’s understanding of the Python programming language and machine learning. It is best to start with the documentation for all imported tools used in development prior to making alterations.