Project Report: ANN MNIST Classification

> **Name :** Narula Vivek

> **Roll Number :** 24/RCO/09 ; **Subject Code** : `RICV505` ; **Group Code: 19**

> Drive Link for this Project

1. Introduction

This project showcases the implementation of an **ANN(Artificial Neural Network)** from scratch in order to classify handwritten digits with the help of a subset of the **MNIST** dataset. The target is to enchance the learning of **ANN** by implementing them by using only standard P**ython** libraries like **numpy**, **scipy** and **matplotlib**. This project also worked as an introduction to the usage of different types of file extensions such as **.h5** files and **.pkl** files which are commonly used to store data in a compressed format.

2. Objective

The primary objectives of this project are:

- To build a custom **ANN** from scratch using **NumPy**.
- Learn the root level working of a neural network by implementing it, as certain libraries and frameworks such as **sklearn** and **pytorch** create a layer of abstraction during this process.
- To compare the performance of different architectures, including **one hidden layer** and **three hidden layers**.
- To check the effect of different activation function such as sigmoid and RELU in this project.
- To utilize Scikit-learn's `MLPClassifier` for comparison.

3. Methodology

3.1 Dataset

For the purpose of this project, we have considered a subset of the **MNIST** dataset. This **MNIST** subset is stored in the **.h5** file format.

Sample size : The dataset has **14251** samples of images in **28 X 28 pixels**. From this we split our dataset into training and test set.

- **Training Set:** 11400 images
- **Test Set:** 2851 images
- The images are also flattened into a dimension of **784** i.e. (28 * 28 pixels)

> 💡 Note : Our dataset consists of images for only the **‘7’** and **‘9’** digits. Therefore, we consider this as a case of **‘binary classification’** rather than **‘multi-class classification’**.

The dataset is loaded and preprocessed for training the **ANN**.

3.2 Data loading

We consider the following steps while loading our data in the `data_loader.py` file:

1. **Step 1 :** Import the **h5py** (A python library to load the .h5 files) and load the data for input images into the variable **X** and their respective labels into the variable **Y.**
2. **Step 2 : Normalize** the pixel values for the input image and **flatten** the image for ease of processing.
3. **Step 3 :** Convert the **labels** into **binary values**. (We map the **digit 7** to **0** and the **digit 9** to **1**).
4. **Step 4 :** Use **80%** of the input data for **training** and **20%** for **testing**.

3.3 Neural Network Architecture

Two architectures were implemented:

1. **Single Hidden Layer Network:**
   - Input Layer: 784 **units** (28x28 pixels)
   - Hidden Layer: 100 **units**
   - Output Layer: 2 **units** (one for each digit)
2. **Three Hidden Layer Network:**
   - Input Layer: 784 **units**
   - Hidden Layers: 100 **units**, 50 **units**, 50 **units**
   - Output Layer: 2 **units** (one for each digit)

3.4 Training Process

The training process involves:

- Forward propagation and using softmax function to compute outputs.
- One-hot encoding of the label vector (as we compute the outputs using softmax).
- Backpropagation to update **weights** and **biases** using the **gradient descent algorithm**.
- Training at **201** epochs with a learning rate of **0.01** in case of using the **RELU** as an activation function.
- Training at **201** epochs with a learning rate of **0.1** in case of using the **Sigmoid** activation function.
- Monitoring training and test loss and accuracy metrics over epochs.
- Save the best weights and biases into our models directory as `best_model_*_hidden_*.pkl` in the `pickle` format.
- Loading the `best_model_*_hidden_*.pkl` into the respective archietecture in order to compute accuracy on the test set.

💡 Additional details for the use of each file are mentioned in the README.md present in the attached GitHub Repository.

4. Results

The performance of the models was evaluated based on accuracy:

Model	Activation Function	Initialization	Hidden Layers	Train Accuracy (%)	Test Accuracy (%)
model_RELU.py	ReLU	He	1 Hidden Layer	93.53	94.46
model_RELU.py	ReLU	He	3 Hidden Layers	94.76	95.69
model_Sigmoid.py	Sigmoid	Xavier	1 Hidden Layer	94.00	94.91
model_Sigmoid.py	Sigmoid	Xavier	3 Hidden Layers	81.17	84.32

4.1 Key Observations:

- **ReLU Activation (model_RELU.py)**:
  - Performs better with both 1 hidden layer (94.46%) and 3 hidden layers (95.69%) using **He initialization**.
  - **Test accuracy improves with more hidden layers**.
- **Sigmoid Activation (model_Sigmoid.py)**:
  - Performs similarly for 1 hidden layer (94.91%) but has **a significant drop in performance** with 3 hidden layers (84.32%) using **Xavier initialization**.
  - **Adding more hidden layers negatively impacts performance**, this can be due to the vanishing gradient problem that may occur in the Sigmoid function.

This comparison highlights that **ReLU** with **He initialization** is generally better for deeper networks, while **Sigmoid** may perform well in shallow networks but struggles with deeper ones.

4.2 Check for Underfitting or Overfitting

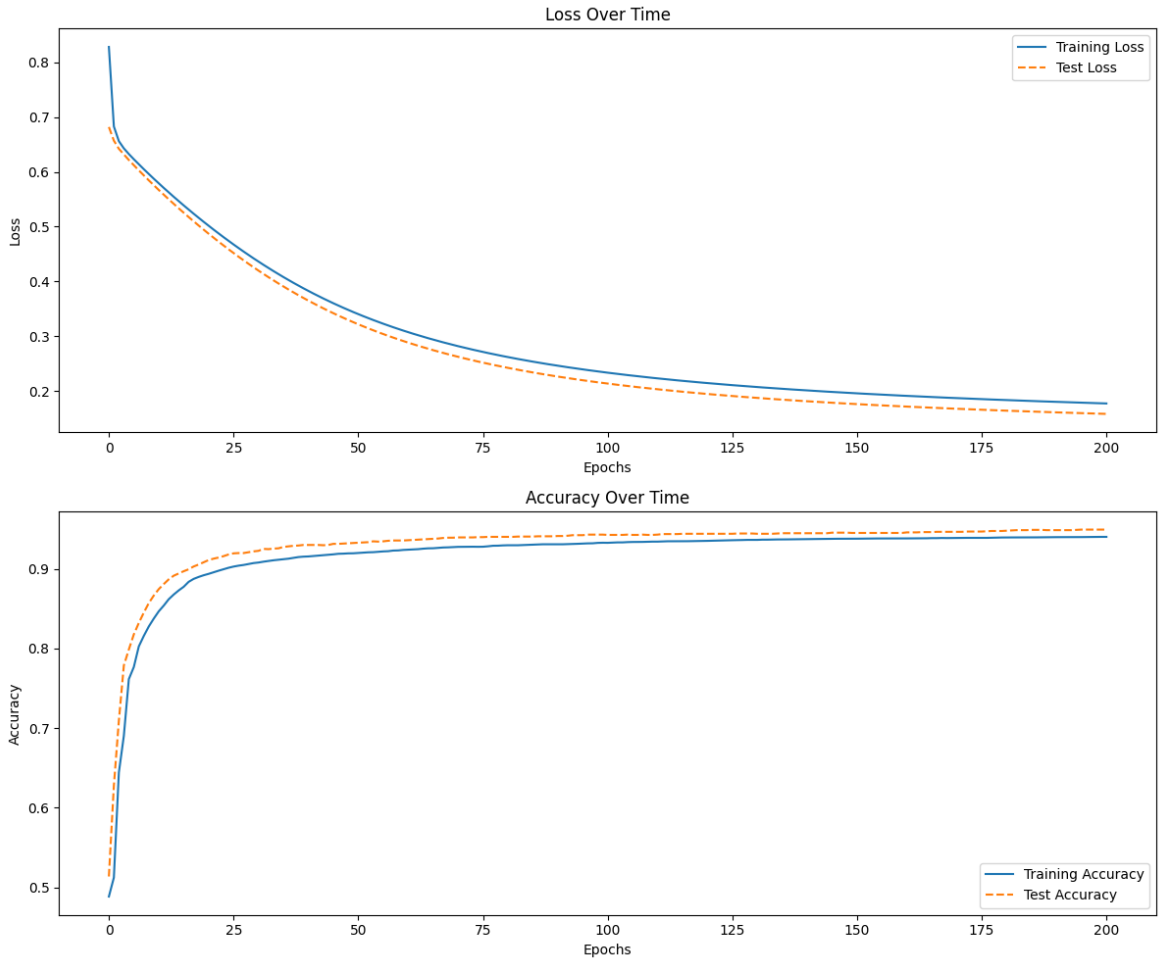

Fig 1 : Shows the loss and accuracy over epochs for the **Single Hidden Layer** network with **sigmoid** activation.

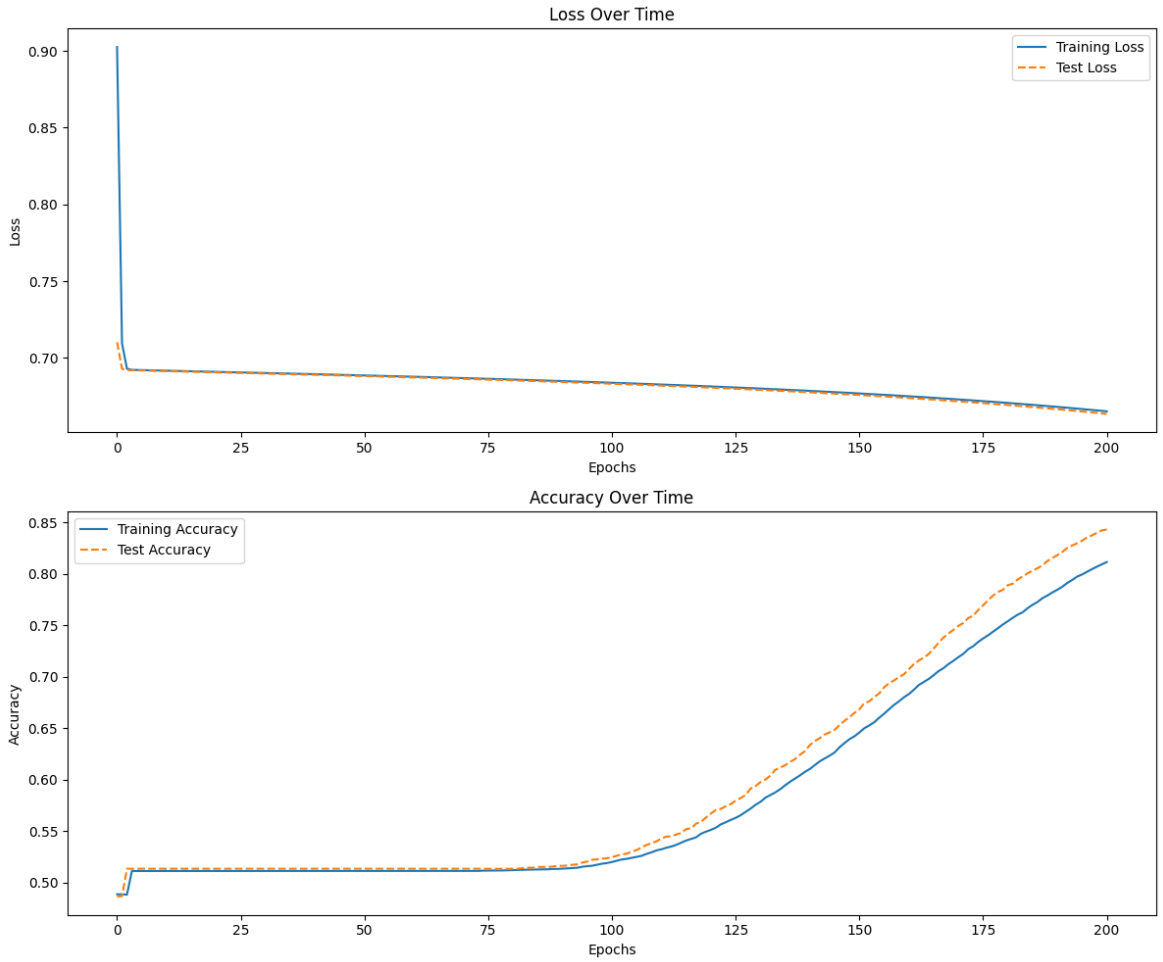

Fig 2 : Shows the loss and accuracy over epochs for the **Three Hidden Layer** network with **sigmoid** activation.

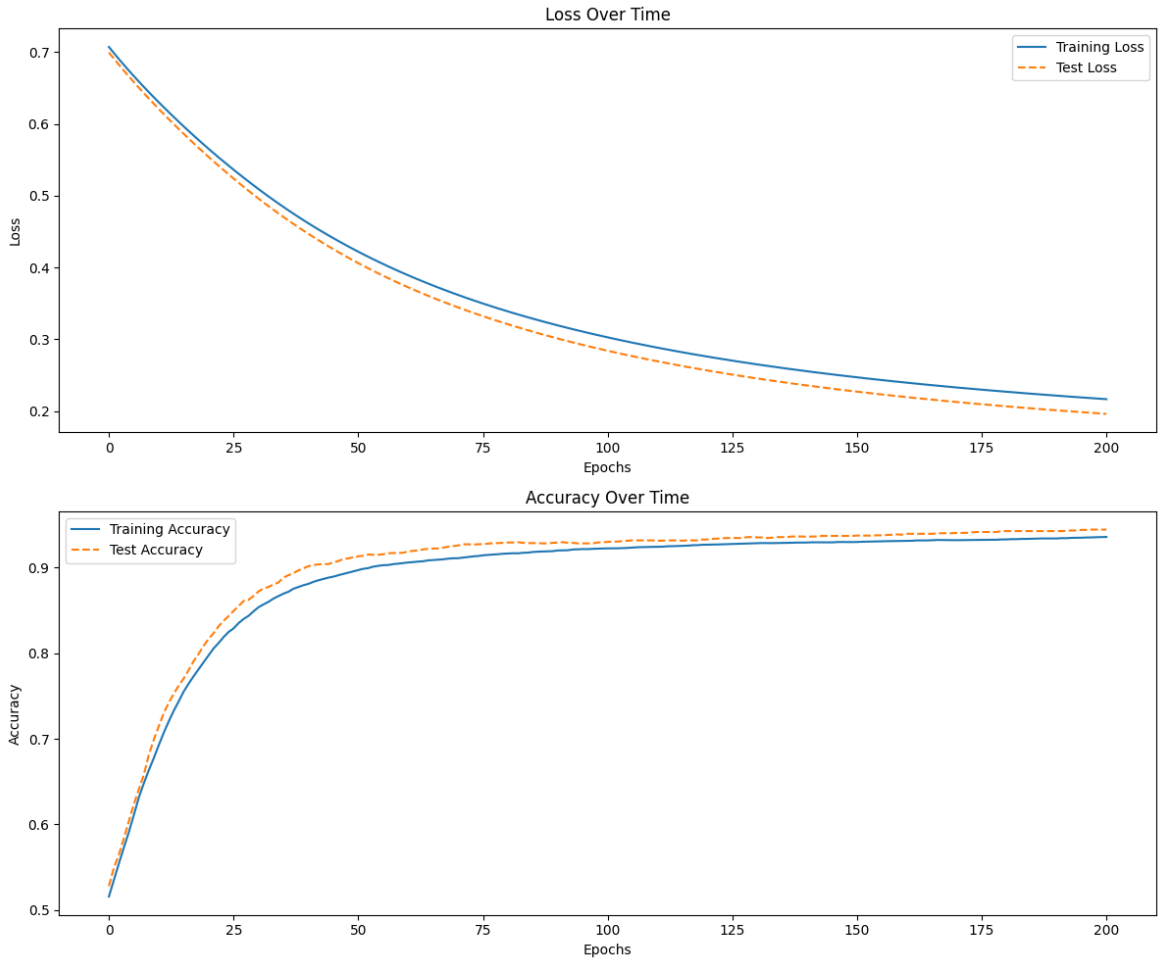

Fig 3 : Shows the loss and accuracy over epochs for the **Single Hidden Layer** network with **RELU** activation.

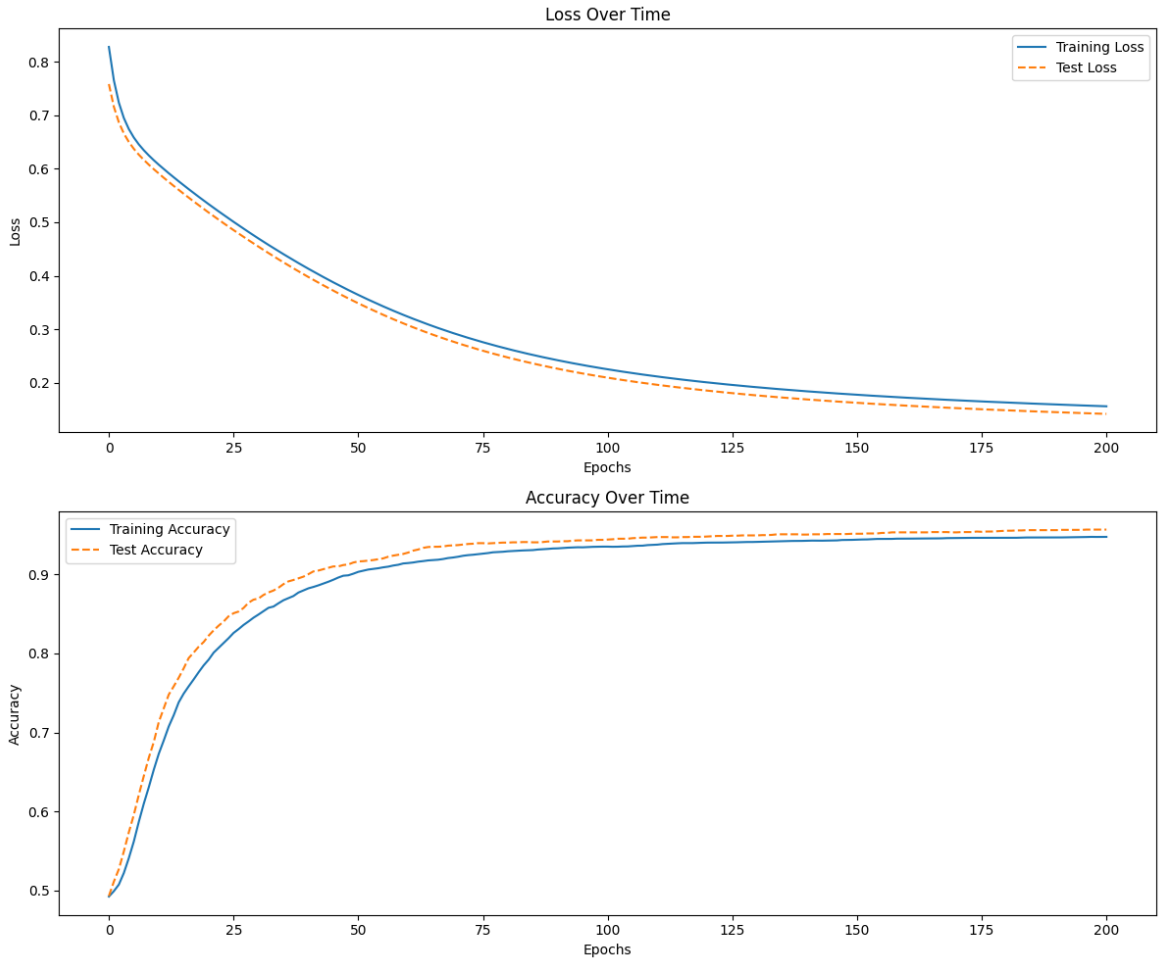

Fig 4 : Shows the loss and accuracy over epochs for the **Three Hidden Layer** network with **RELU** activation.

`Matplotlib was used in order to produce the above plots while training and testing.`

4.2.1 Conclusion:

As our **loss** decreases over **epochs** and also our **accuracy increases** over **epochs** for both the **Training** and **Test set**, we conclude that our model staisfies the requirement of a **good model** since it is neither **overfitting** nor **underfitting**.

NOTE: This holds true for all of the 4 models designed as part of this project and the same can be seen in the above **Fig (1 to 4)**

4.3 Challenges faced and counter-measures taken:

1. All of the designed models did not perform well with random initialization of weights and biases, therefore we considered **He** initialization for **RELU** activation function and **Xavier** initialization for the **sigmoid** activation function.
2. All of the designed models do not see a major decrease in loss or increase in accuracy after **200 epochs** therefore, we stop training at **201** epochs where **201** is considered just for the sake of better representation.
3. Another issue was the loading of the dataset as it was provided with the .**h5** format and therefore a popular **Python** library called `h5py` was used to load the dataset and also **Python's** default **indexing** and **slicing**

operations were used to separate the data into **training** and **testing** sets. The detailed procedure is present in the `data_loader.py` file.

4. We made use of the `pickle` library in **Python** in order to store the values of our best available **weights** and **biases** for each model which were obtained after training the model. This was done in order to prevent re-training our model each time and it enables us to use these **weights** and **biases** in the future thereby reducing the computation cost.

4.4 Comparison with the scikit-learn implementation

Model Performance Comparison

Model	Activation Function	Initialization	Hidden Layers	Train Accuracy (%)	Test Accuracy (%)
model_RELU.py	ReLU	He	1 Hidden Layer	94.00	94.46
model_RELU.py	ReLU	He	3 Hidden Layers	94.76	95.69
model_Sigmoid.py	Sigmoid	Xavier	1 Hidden Layer	94.00	94.91
model_Sigmoid.py	Sigmoid	Xavier	3 Hidden Layers	81.17	84.32
Scikit-learn MLP	ReLU	(N/A)	1 Hidden Layer	(N/A)	**99.30**
Scikit-learn MLP	ReLU	(N/A)	3 Hidden Layers	(N/A)	**99.33**

4.4.1 Key Observations and reasoning:

Here, we see that the **scikit-learn's MLPClassifier** outperforms our previous models, the possible reasons can be as follows:

1. **The usage of more iterations to train:** The **sklearn's MLPClassifier** in our code used **500** iterations to train the model whereas we only used 200 epochs while training our models designed with **numpy.**
2. **The usage of better optimizer:** The **sklearn's MLPClassifier** in our code used the **Adam** optimizer rather than the **Gradient descent** that was being used in the training of our previous models.

4.4.2 Observations

- Both the custom **ANN** and **Scikit-learn** implementations achieved high accuracy, demonstrating the effectiveness of the chosen architectures.
- The three hidden layer architecture performed best, achieving an accuracy of 99.23% when using the **scikit-learn's MLPClassifier.**
- The **three hidden layer** architecture on the sigmoid activation function performed the worst with an accuracy of **84.32%**

5. Conclusion

This project succesfully completed all it's objectives and lead to the enhanced and in-depth learning of ANNs. With this project we also saw the various methods of importing and loading our data as well as storing it in different file formats. Though the data provided was a subset of **MNIST,** the model designed are still sufficient enough to be trained on the whole of the MNIST dataset and lead to a highly accurate result.

6. Future Work

Future improvements can focus on:

- Application of different types of neural networks such as CNNs in order imporve accuracy and efficiency.
- Creating a parser in order to enable the usage of the code from the command line.

6.1 Additional Information

The project details such as the information regarding each of the code file and the file system architecture as well as the information on the execution of the code and the command to install all the libraries used in this project are all available in the **GitHub** repository for this project which will be attached in the references section below. The repository also contains the table of comparison among various models.

> 💡 The repostiory will only be made public after the due date for the submission of this project has passed.

References

1. **NumPy**: A fundamental package for numerical computing in Python.
   - Documentation: NumPy Documentation
   - GitHub Repository: NumPy GitHub
2. **SciPy**: A library used for scientific and technical computing.
   - Documentation: SciPy Documentation
   - GitHub Repository: SciPy GitHub
3. **Matplotlib**: A plotting library for creating static, animated, and interactive visualizations in Python.
   - Documentation: Matplotlib Documentation
   - GitHub Repository: Matplotlib GitHub
4. **Scikit-learn**: A machine learning library for Python, offering simple and efficient tools for data mining and data analysis.
   - Documentation: Scikit-learn Documentation
   - GitHub Repository: Scikit-learn GitHub
5. **h5py**: A Pythonic interface to the HDF5 binary data format, useful for storing large amounts of data.
   - Documentation: h5py Documentation
   - GitHub Repository: h5py GitHub
6. **Pickle**: A Python module for serializing and de-serializing Python object structures.
   - Documentation: Pickle Documentation
7. **MNIST Dataset**: A large database of handwritten digits, commonly used for training various image processing systems.

- Dataset Overview: MNIST Database

8. **Python**: The programming language used for this project.
   - Official Website: Python.org
   - Documentation: Python Documentation
9. **Additional Resources**:
   - "Neural Networks and Deep Learning" by Michael Nielsen: Book Website
   - "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Book Website
10. **GitHub Link**
    - GitHub Repository for the Project