
CSE489 Selected Topics in Data Science – Natural Language Processing

# Project Documentation

**Submitted to:**

Prof. Dr. Alaa Hamdy

**Submitted by:**

Anthony Amgad Fayek 19P9880

Mahmoud Mohamed Elnashar 19P3374

# Next Word Predictor

# Table of Contents

# 1. Introduction

The Next Word Predictor project is a machine learning application that aims to predict the subsequent word in a sequence of text. Such predictive text models have wide-ranging applications, from enhancing user experience in mobile and desktop typing to aiding in natural language understanding tasks. This project leverages advanced neural network architectures, including Convolutional Neural Networks (CNNs), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs), to build a robust next word predictor.

Initially, the model was designed to handle input sequences of 25 words, providing a comprehensive context for the prediction task. However, it was observed that such long sequences increased computational complexity and processing time. To address this, the input sequence length was reduced to 5 words. This reduction aimed to strike a balance between maintaining sufficient context for accurate predictions and optimizing computational efficiency.

The choice of neural network architectures is critical in handling sequential data. CNNs, while typically known for their success in image processing, have been explored for text processing due to their ability to capture local dependencies and patterns in data. However, in this project, CNNs did not perform as well as expected for the next word prediction task. Their ability to model sequential dependencies was limited compared to LSTMs and GRUs, which are specifically designed to handle such sequential data by maintaining long-term dependencies and mitigating issues related to vanishing and exploding gradients.

A significant part of this project involved extensive preprocessing steps to prepare the text data for training. This included tokenization and embedding. Tokenization breaks down the text into individual tokens (words) and embedding maps these tokens into high-dimensional vectors. These steps are crucial for feeding the data into neural network models effectively.

Overall, this project not only highlights the potential of deep learning techniques in text prediction tasks but also provides insights into the practical challenges and considerations involved in building and optimizing such models.

# 2. Dataset

In this project, three different datasets were considered for training and evaluating the next word predictor model, each with distinct characteristics and challenges.

## 2.1. Project Gutenberg's The Adventures of Sherlock Holmes by Arthur Conan Doyle

The first dataset consisted of the full text from "The Adventures of Sherlock Holmes," a classic piece of literature available from Project Gutenberg. This dataset was robust and extensive, providing a rich source of text. However, the language used in the book is somewhat archaic and contains many old English words and expressions that are not commonly used in contemporary language. As a result, it did not align well with the project's goal of predicting modern text sequences, leading to its exclusion despite its potential richness.

## 2.2. News Headings

The second dataset was composed of various news headlines. This dataset provided a different approach, offering shorter, more concise text samples. While it presented a more modern vocabulary compared to the Sherlock Holmes dataset, it was predominantly uniform in nature. The lack of variety in sentence structures and themes meant that the text was too homogeneous, limiting the model's ability to generalize across different types of language. Consequently, the news headings dataset did not prove effective for the intended use case.

## 2.3. Synthetic Phone Call Conversations

The third dataset consisted of synthetic phone call conversations, which ultimately proved to be the best fit for this project. This dataset was designed to mimic the varied nature of real-world phone conversations, incorporating both formal and informal language. The diversity in conversational styles and topics provided a rich training ground for the model, enabling it to learn from a broad spectrum of language patterns. The synthetic nature of the dataset ensured that it was comprehensive and tailored to the needs of predictive text tasks, making it the most suitable choice for developing a versatile next word predictor.

By experimenting with these three datasets, it was evident that the synthetic phone call conversations dataset offered the best balance of modern language usage and variety, crucial for training an effective next word prediction model.

# 3. Preprocessing

The preprocessing step is crucial for transforming raw text data into a format suitable for training neural network models. The following steps outline the preprocessing pipeline used in this project:

## 3.1. Tokenization

Tokenization is the process of breaking down text into individual units, such as words or tokens. In this project, the text was split into tokens using spaces as delimiters.

```python
tokens = text.split(" ")
tokens = list(filter(lambda t: t != "", tokens))
tokens
```

```
['Hi',
 'this',
 'is',
 'the',
 'systems',
 'development',
 'department',
```

Here, the text is split by spaces into a list of tokens. Any empty strings resulting from multiple spaces are then removed using the filter function.

## 3.2. Sequence Generation

After tokenization, the text was converted into sequences of a fixed length to create training samples. Each sequence consists of train_len words, where the first train_len - 1 words serve as input, and the last word is the target output.

```python
train_len = 5+1
text_sequences = []
for i in range(train_len, len(tokens)):
    seq = tokens[i-train_len:i]
    text_sequences.append(seq)
```

In this step, a sliding window approach is used to generate sequences. The train_len is set to 6, meaning each sequence will contain 5 words for input and 1 word for the target prediction. This method ensures that the model learns to predict the next word based on the previous 5 words.

## 3.3. Text Tokenization

The sequences generated were then tokenized using Keras' Tokenizer class, which converts the text sequences into numerical values (indices) that represent each unique word.

```python
tokenizer = Tokenizer()
tokenizer.fit_on_texts(text_sequences)
sequences = tokenizer.texts_to_sequences(text_sequences)
```

The Tokenizer builds a vocabulary of all unique words in the text sequences and assigns each word a unique integer index. This numerical representation of words is necessary for feeding the data into neural network models.

## 3.4. Embedding

To convert tokens into a numerical format that the neural network can process, an embedding layer was used. Embeddings map each token to a high-dimensional vector, capturing semantic relationships between words.

```
model.add(Embedding(vocabulary_size, 50, input_length=5))
```

The embedding layer creates a dense representation of the words, with an embedding dimension of 50. The input length is set to 5, corresponding to the number of words in the input sequence.

By following these preprocessing steps, the text data was effectively prepared for training the neural network models. The tokenization and sequence generation steps ensured that the data was in the right format, while embedding allowed the model to process the sequences efficiently. This comprehensive preprocessing pipeline was essential for building a robust and accurate next word predictor.

# 4. Models

## 4.1. CNN

The convolutional neural network (CNN) model designed for next word prediction leverages various layers to process the input text sequences and predict the next word. Here is an in-depth explanation of the architecture and functionality of the model:

### 4.1.1. Embedding Layer

The first layer in the model is an Embedding layer, which converts the input tokens into dense vectors of fixed size. This layer helps in capturing the semantic relationships between words.

- **Input:** The input length is 5, corresponding to the 5-word sequences generated during preprocessing.
- **Output:** Each word is represented as a 50-dimensional vector, enabling the model to learn richer representations of the input text.

### 4.1.2. Dense Layers

The model includes two Dense (fully connected) layers, each with ReLU activation. These layers help in learning complex patterns and relationships in the data by transforming the input features.

- **First Dense Layer:** Contains 64 units, adding non-linearity to the model and allowing it to learn more intricate patterns.
- **Second Dense Layer:** Contains 32 units, further refining the learned features and preparing them for the convolutional layer.

### 4.1.3. Convolutional Layer

The Conv1D layer applies a one-dimensional convolutional operation over the input sequence, which helps in capturing local dependencies and patterns in the text data.

- **Filters:** The convolutional layer uses 32 filters.
- **Kernel Size:** A kernel size of 1 is used, meaning the convolution operation considers one word at a time.

- **Activation:** ReLU activation introduces non-linearity and helps in learning complex patterns in the input data.

#### 4.1.4. Global Max Pooling Layer

This layer performs a max pooling operation over the output of the convolutional layer, reducing the dimensionality and retaining the most important features.

- **Operation:** Takes the maximum value over the time dimension, effectively summarizing the most significant features detected by the convolutional layer.

#### 4.1.5. Output Layer

The final layer is a Dense layer with softmax activation, which outputs a probability distribution over the vocabulary, predicting the likelihood of each word being the next word in the sequence.

- **Units:** The number of units equals the vocabulary size, ensuring that the model outputs a probability for each word in the vocabulary.
- **Activation:** Softmax activation converts the logits into probabilities, making it suitable for multi-class classification.
- **Regularization:** L2 regularization is applied to prevent overfitting by penalizing large weights.

#### 4.1.6. Compilation

The model is compiled with categorical cross-entropy loss, which is appropriate for multi-class classification tasks. The Adam optimizer is used for efficient gradient descent.

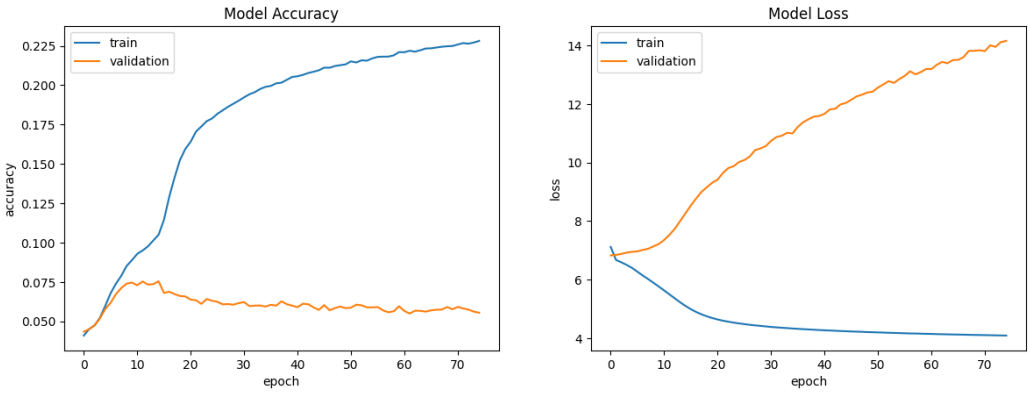
- **Loss Function:** Categorical cross-entropy measures the difference between the predicted and actual distributions.
- **Optimizer:** Adam optimizer is used for its adaptive learning rate and efficient convergence.
- **Metrics:** Accuracy is used as a metric to evaluate the model's performance during training and validation.

### 4.1.7. Training

The training process involved iterating through the dataset for 75 epochs. Each epoch represents one complete pass through the entire training dataset. Below is a summary of the training performance:

- **Epoch 1:** The initial training loss was 7.1202, with an accuracy of 0.0410. The validation loss was 6.8294, with an accuracy of 0.0434.
- **Epochs 2-75:** Over successive epochs, the training loss gradually decreased, and the training accuracy improved. However, the validation loss continued to increase after certain epochs, indicating potential overfitting. By the final epoch, the training accuracy reached 0.2281, while the validation accuracy was 0.0555.

Despite the improvements in training accuracy, the increasing validation loss and relatively low validation accuracy suggest that the model might benefit from further tuning, such as adjusting the network architecture, experimenting with different regularization techniques, or exploring more diverse and larger datasets.




### 4.1.8. Code

```
model = Sequential()
model.add(Embedding(vocabulary_size, 50, input_length=5))
model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=32, activation='relu'))
model.add(Conv1D(32,1,activation="relu"))
model.add(GlobalMaxPooling1D())
model.add(Dense(vocabulary_size, activation='softmax', activity_regularizer=regularizers.L2(0.001)))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

## 4.2. GRU

The Gated Recurrent Unit (GRU) model designed for sequence-based tasks, such as next word prediction or text generation, utilizes several layers to process input sequences and make predictions. Here's a detailed explanation of the architecture and functionality of the model:

### 4.2.1. Embedding Layer

The first layer in the model is an Embedding layer, which transforms input word indices into dense vectors of fixed size.

- Input: The vocabulary size and input length are specified, with each word represented by a 50-dimensional vector.
- Output: The embedding layer converts the sparse word representations into continuous, dense vectors, aiding in capturing semantic relationships between words.

### 4.2.2. Bidirectional GRU Layer

The model includes a Bidirectional GRU layer with 100 units, operating on the embedded input sequences in both forward and backward directions.

- Operation: This layer captures context from past and future states relative to each time step, effectively encoding temporal dependencies in the input sequences.
- Return Sequences: The parameter return_sequences=True ensures that the output at each time step is returned, preserving sequence information for subsequent layers.

### 4.2.3. Dropout Layer

To prevent overfitting, a Dropout layer with a dropout rate of 0.2 is added after the bidirectional GRU layer.

- Operation: Dropout randomly sets a fraction of input units to zero during training, promoting robustness and reducing the likelihood of overfitting.

### 4.2.4. GRU Layer

Another GRU layer with 100 units follows the dropout layer, processing the sequence output from the bidirectional GRU layer.

- Operation: This GRU layer summarizes the sequence information into a fixed-length vector, capturing the most salient features from the entire sequence.

- Output: Since `return_sequences` is not specified (default is `False`), this layer only returns the output for the last time step.

#### 4.2.5. Dense Layers

The model includes a Dense layer with 100 units and ReLU activation, serving as an intermediate layer to learn complex representations.

- Operation: This layer helps in capturing higher-level abstractions and patterns from the features extracted by the GRU layers.

#### 4.2.6. Output Layer

The final Dense layer outputs a probability distribution over the vocabulary, predicting the likelihood of each word being the next word in the sequence.

- Units: The number of units equals the vocabulary size, ensuring the model outputs a probability for each word.
- Activation: Softmax activation converts the logits into probabilities, facilitating multi-class classification.
- Regularization: L2 regularization with a coefficient of 0.001 is applied to prevent overfitting by penalizing large weights.

#### 4.2.7. Compilation

The model is compiled with categorical cross-entropy loss and the Adam optimizer.

- Loss Function: Categorical cross-entropy measures the difference between predicted and actual distributions.
- Optimizer: Adam optimizer updates the network weights efficiently during training, with adaptive learning rate capabilities.
- Metrics: Accuracy is used to evaluate the model's performance during training and validation.

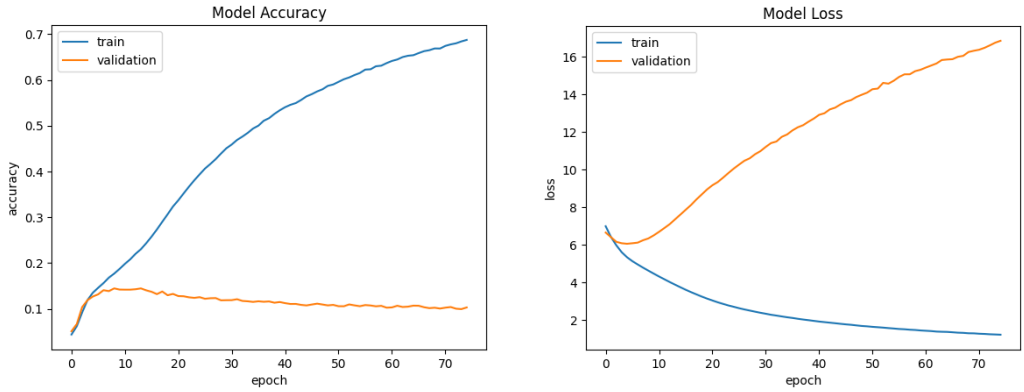
#### 4.2.8. Training

During training, the model iterates through the dataset for 75 epochs. Here's a summary of the training performance:

- Epoch 1: The initial training loss was 6.9710, with an accuracy of 0.0437. The validation loss was 6.6352, with an accuracy of 0.0510.

- Epochs 2-75: The training loss gradually decreased, and the accuracy improved over successive epochs. However, the validation loss started to increase after a certain point, indicating potential overfitting.

By the final epoch, the training accuracy reached 0.6874, while the validation accuracy was 0.1029. Despite improvements in training accuracy, the increasing validation loss suggests that the model might benefit from further tuning to improve generalization performance.




#### 4.2.9. Code

```
model = Sequential()
model.add(Embedding(vocabulary_size, 50, input_length=5))
model.add(Bidirectional(GRU(100, return_sequences=True)))
model.add(Dropout(0.2))
model.add(GRU(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocabulary_size, activation='softmax', activity_regularizer=regularizers.L2(0.001)))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

## 4.3. LSTM

The Long Short-Term Memory (LSTM) model, constructed with a sequential architecture, serves the purpose of sequence prediction, notably for tasks like next word prediction or text generation. The model comprises several key layers, each contributing to its overall functionality:

#### 4.3.1. Embedding Layer

The initial layer in the LSTM model is the Embedding layer, tasked with transforming input word indices into dense vectors of fixed size. This transformation facilitates the capturing of semantic relationships between words within the input sequences.

- Input: The input length is 5, corresponding to the 5-word sequences generated during preprocessing.
- Output: Each word is represented as a 50-dimensional vector, enabling the model to learn richer representations of the input text.

#### 4.3.2. LSTM Layers

The model incorporates two LSTM layers, each with distinct roles in processing the input sequences and capturing temporal dependencies:

- First LSTM Layer: Comprising 150 units and configured to return sequences, this layer captures temporal dependencies bidirectionally, enhancing the model's understanding of context.
- Second LSTM Layer: Also consisting of 150 units, this layer further refines the learned representations from the previous layer, summarizing sequence information into a fixed-length vector.

#### 4.3.3. Dense Layer

Following the LSTM layers, a Dense layer with 150 units and ReLU activation is employed to learn complex patterns and relationships in the data, preparing the model for the final prediction step.

#### 4.3.4. Output Layer

The output layer, a Dense layer with softmax activation, generates a probability distribution over the vocabulary, predicting the likelihood of each word being the next in the sequence.

- Regularization: L2 regularization with a coefficient of 0.001 is applied to prevent overfitting by penalizing large weights.

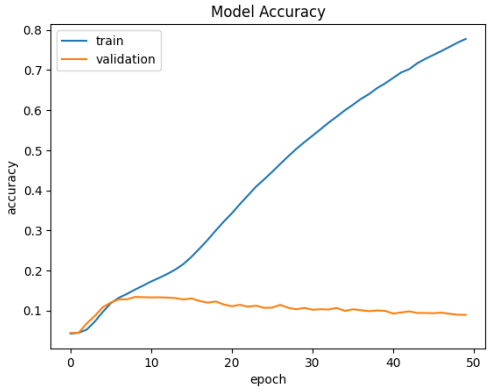
#### 4.3.5. Compilation

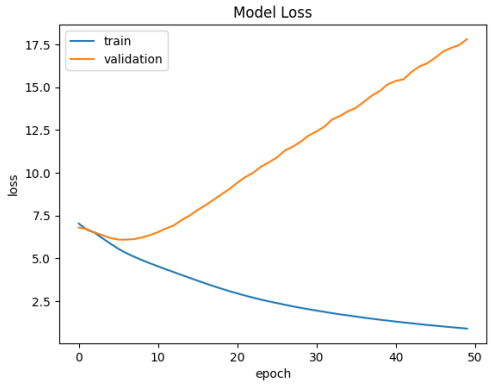
During compilation, the model is configured with categorical cross-entropy loss and the Adam optimizer, which efficiently updates network weights based on computed gradients.

- Loss Function: Categorical cross-entropy measures the difference between predicted and actual distributions.
- Optimizer: The Adam optimizer facilitates efficient gradient descent with adaptive learning rates.

### 4.3.6. Training

The training process involves iterating through the dataset for 50 epochs, with both training and validation losses gradually decreasing over successive epochs, indicating the model's learning progress. However, the validation accuracy remains relatively low, suggesting potential areas for further optimization or architectural adjustments to enhance generalization performance.




### 4.3.7. Code

```
model = Sequential()
model.add(Embedding(vocabulary_size, 50, input_length=5))
model.add(LSTM(150, return_sequences=True))
model.add(LSTM(150))
model.add(Dense(150, activation='relu'))
model.add(Dense(vocabulary_size, activation='softmax', activity_regularizer=regularizers.L2(0.001)))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

## 4.4. Comparison

- CNN: Trained for 75 epochs, with the training accuracy reaching 0.2281 and the validation accuracy at 0.0555. The model exhibited signs of potential overfitting, with increasing validation loss and relatively low validation accuracy.
- LSTM: Trained for 50 epochs, with the training accuracy improving gradually to 0.7778, while the validation accuracy remained relatively low at 0.0894. The model demonstrated a better ability to capture temporal dependencies and long-range context compared to CNN.
- GRU: Trained for 50 epochs, with the training and validation accuracies reaching 0.6874 and 0.1029, respectively. Like LSTM, GRU showed promising results in learning sequential patterns and dependencies while being computationally more efficient.

In summary, while CNNs excel at capturing spatial patterns, LSTMs and GRUs are better suited for tasks requiring understanding of temporal dependencies and long-range context. LSTMs offer a balance between complexity and performance, while GRUs provide similar capabilities with reduced computational overhead. The choice between these models depends on the specific requirements of the task and considerations regarding computational resources and performance metrics.

The following table shoes the result of each model when the phrase ***"The quick brown fox jumps"*** is entered and 3 words are predicted recursively:

| | |
|---|---|
| CNN | "spots classes the" |
| GRU | "away by the" |
| LSTM | "explain me how" |

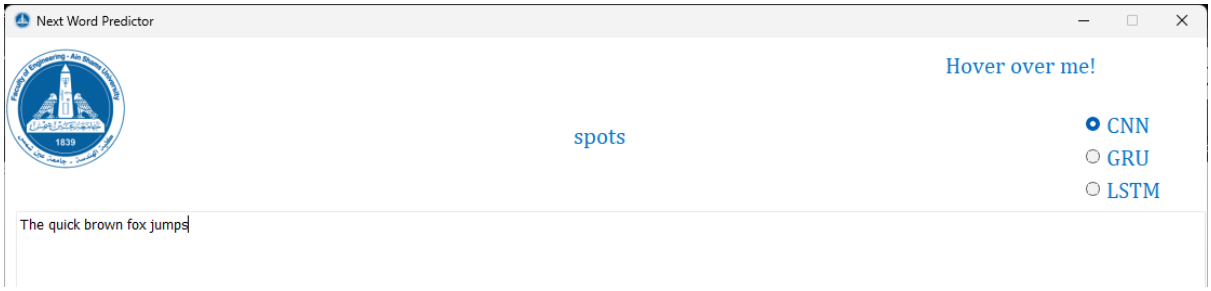
# 5. Python App

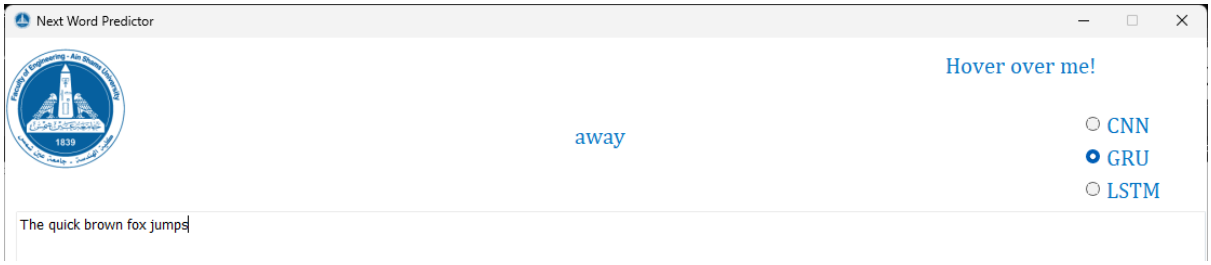
A Python application was developed using PyQt to demonstrate the functionality of the next word prediction models in real-time. The application provides a user interface where users can input text, and the models predict the next word based on the input sequence.

The application utilizes the following components:

- Graphical User Interface (GUI): Developed using PyQt, the GUI allows users to interact with the application easily. Users can input text and view the predicted next word.
- Next Word Prediction Models: The application integrates Convolutional Neural Network (CNN), Gated Recurrent Unit (GRU), and Long Short-Term Memory (LSTM) models for next word prediction. These models were trained on text data and are capable of predicting the next word in a sequence based on the input text provided by the user.
- Model Loading: The application loads pre-trained models from JSON and h5 files. These files contain the architecture and weights of the trained models, enabling the application to use them for prediction.
- Tokenization: Before making predictions, the input text is tokenized using tokenizer objects saved during model training. This tokenization process converts the text into sequences of integers, which are then used as input to the models.
- Prediction: After tokenization, the models predict the next word in the sequence based on the input text provided by the user. The predicted word is displayed in the GUI, allowing users to see the model's output in real-time.

The Python application enhances user experience by providing a visual representation of the next word prediction models' capabilities. Users can experiment with different input texts and observe how the models predict the next word, gaining insights into the models' behavior and performance.


Next Word Predictor
Hover over me!
spots
CNN
GRU
LSTM
The quick brown fox jumps


Next Word Predictor
Hover over me!
away
CNN
GRU
LSTM
The quick brown fox jumps


Next Word Predictor
Hover over me!
explain
CNN
GRU
LSTM
The quick brown fox jumps

# 6. Important Links and Information

*GitHub Repository*

*Executable Download*

This build was created using Pyinstaller and the source code as well as the spec file can be found under the directory ‘Project Program’ in the GitHub repository.

*Dataset*

Here’s a list of the packages required to run the notebooks:

- `keras`
- `tensorflow`
- `numpy`
- `seaborn`
- `pandas`
- `matplotlib`
- `opencv-python`
- `scikit-learn`

Here’s a list of the packages required to run the python program file:

(THE EXE DOESN’T REQUIRE ANY PREREQUISITES TO RUN)

- `keras`
- `tensorflow`
- `numpy`