Department of Computer Science

University of Delhi

BONE AGE PREDICTION

PROJECT REPORT

Subject: Data Science (MCSO302)

Submitted By:

Bharat Chudasama (14)
Rudra Bhardwaj (36)
Ankit Yadav (09)
Sawan Bagle (39)
Tushar Gaur (48)

Submitted To:

Dr. Bharti Rana

November 4, 2025

Abstract

This report details the development and evaluation of machine learning models for the regression task of predicting bone age from hand X-ray images. The primary objective is to build a robust model and compare the efficacy of different deep learning architectures and ensemble techniques. This project implements four distinct models: (1) a custom Convolutional Neural Network (CNN) built from scratch, (2) a transfer learning model using the VGG16 architecture with frozen base layers, (3) a weighted ensemble combining the predictions of the base models, and (4) a stacking ensemble using a `LinearRegression` meta-model. Both base models were optimized using Keras Tuner. All models were evaluated on a held-out test set using R2 Score, Mean Absolute Error (MAE), Mean SquaredError (MSE), and Root Mean Squared Error (RMSE). The results demonstrate that the Stacking Ensemble model achieved the superior performance, yielding an R2 Score of 0.7233 and an RMSE of 22.1370 months, indicating its strong potential for this medical imaging task.

Acknowledgement

We would like to express our sincere gratitude to our professor, **Dr. Bharti Rana**, for her invaluable guidance, patience, and support throughout this project. Her expertise and constructive feedback were instrumental in shaping our research and overcoming challenges.

We would also like to thank the **Department of Computer Science, University of Delhi**, for providing the necessary resources and a conducive learning environment to complete this work.

Finally, we are grateful to our batchmates for the insightful discussions and collaborative spirit.

Bharat Chudasama (14)

Rudra Bhardwaj (36)

Ankit Yadav (09)

Sawan Bagle (39)

Tushar Gaur (48)

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Statement

Skeletal maturity, commonly referred to as "bone age," is a critical diagnostic indicator in pediatrics. It provides a measure of a child's development that is more indicative of their biological maturity than their chronological age. Assessing bone age is essential for diagnosing a variety of endocrine and metabolic disorders that can affect growth, such as growth hormone deficiency, hypothyroidism, and precocious puberty. It is also vital for predicting a child's final adult height and for timing orthopedic interventions. The traditional method for bone age assessment, such as the Greulich-Pyle (GP) atlas, involves a subjective comparison of a patient's hand X-ray to a reference atlas, a process that is time-consuming and suffers from significant inter-observer variability.

1.2 Project Goal

The primary objective of this project is to automate the bone age assessment process by developing a deep learning-based regression model. The goal is to accurately predict the bone age in months from a given hand X-ray image. To achieve this, this project explores and rigorously compares the performance of several models:

- A custom Convolutional Neural Network (CNN) tailored for this task.
- A pre-trained VGG16 model adapted using transfer learning.
- Two ensemble methods (weighted and stacking) that combine the predictions of the base models to potentially achieve superior accuracy.

1.3 Report Structure

This report is organized as follows: Section 2 discusses the dataset and the preprocessing steps taken to prepare the images for modeling. Section 3 details the architecture, training, and hyperparameter tuning process for each of the models. Section 4 presents the final evaluation results, comparing the models using several key regression metrics. Section 5 provides a discussion of these results, and Section 6 concludes the report with a summary of findings and suggestions for future work.

2 Dataset and Preprocessing

2.1 Dataset

The project utilizes the "Bone Age Training Dataset," a publicly available collection of hand X-ray images. The dataset consists of two main components:

- `boneage-training-dataset.csv`: A CSV file mapping each image `id` to its corresponding `boneage` (the target variable, in months) and the patient's sex.
- `boneage-training-dataset/`: A directory containing thousands of X-ray images in `.png` format.

This rich dataset provides the foundation for training and evaluating the regression models.

2.2 Environment Setup

The project was developed in Python, leveraging a suite of powerful libraries for data manipulation, machine learning, and deep learning.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.applications import VGG16
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
import keras_tuner as kt
```

2.3 Data Preprocessing

A critical step in any computer vision task is to preprocess the images into a uniform format that the neural network can accept.

2.3.1 Image Loading and Normalization

A custom function, `load_images`, was implemented to process the data. This function performs the following steps for each image:

1. **Load Image**: Reads the image file from disk.
2. **Resize**: Resizes every image to a uniform dimension of **(224, 224)** pixels. This is a common input size for many pre-trained models, including VGG16.
3. **Convert to Array**: Converts the image to a NumPy array.
4. **Normalize**: Scales the pixel values from the range [0, 255] to [0.0, 1.0] by dividing by 255.0. Normalization is crucial for stabilizing and accelerating the training process.

```
IMG_SIZE = (224, 224) # Define image size constant

def load_images(df, path, img_size):
    images = []
    # ... (code for checking missing files) ...
    df_filtered = df.copy()

    for img_name in df_filtered['Image Path']:
        try:
            img = load_img(os.path.join(path, img_name),
                           target_size=img_size)
            img = img_to_array(img) / 255.0
            images.append(img)
        except Exception as e:
            print(f"Error loading image {img_name}: {e}")
            continue

    y_values = df_filtered['boneage'].values
    return np.array(images), y_values, df_filtered
```

2.3.2 Data Splitting

To properly train and evaluate the models, the entire dataset (12,611 images) was divided into three distinct, non-overlapping sets:

- **Training Set (64% - 8070 images)**: Used to fit the parameters of the models.
- **Validation Set (16% - 2018 images)**: Used to tune hyperparameters (via Keras Tuner) and to train the meta-model in the stacking ensemble. This set provides an unbiased estimate of model skill during training.
- **Test Set (20% - 2523 images)**: A completely held-out set used for the final, unbiased evaluation of all trained models.

```
# First split: 80% for train+validation, 20% for test
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Second split: Split the 80% into 80% train (64% overall)
# and 20% validation (16% overall)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.2, random_state=42
)
```

3 Model Architectures and Training

Four distinct modeling approaches were implemented and compared. Both base models (CNN and VGG16) were optimized using Keras Tuner with `RandomSearch` to find the best-performing hyperparameters. An `EarlyStopping` callback was used to prevent overfitting by stopping the training process if the validation loss did not improve for 3 consecutive epochs.

3.1 Model 1: Custom Convolutional Neural Network (CNN)

3.1.1 Architecture

A custom CNN was built from scratch using the Keras Sequential API. The architecture was designed to be flexible, allowing Keras Tuner to explore different configurations of filters and dense units. The general structure consists of:

- 2-3 blocks of `Conv2D` followed by `MaxPooling2D` layers to extract hierarchical features.
- A `Flatten` layer to convert the 2D feature maps into a 1D vector.
- A `Dense` hidden layer with ReLU activation.
- A `Dropout` layer to reduce overfitting.
- A final `Dense` output layer with one unit and a `linear` activation function, appropriate for a regression task.

3.1.2 Hyperparameter Tuning

The `build_cnn_model` function defined the search space for Keras Tuner.

```
def build_cnn_model(hp):
    model = Sequential()
    model.add(Conv2D(
        filters=hp.Int('conv_1_filters', min_value=32, max_value=64, step=32),
        kernel_size=(3,3), activation='relu',
        input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)
    ))
    model.add(MaxPooling2D(2,2))
    # ... (Additional Conv/Pool layers) ...
    if hp.Boolean("add_extra_conv_layer"):
        # ... (Defines 3rd conv layer) ...

    model.add(Flatten())
    model.add(Dense(
        units=hp.Int('dense_units', min_value=256, max_value=512, step=128),
        activation='relu'
    ))
    model.add(Dropout(
        rate=hp.Float('dropout_1', min_value=0.1, max_value=0.5, step=0.1)
    ))
    model.add(Dense(1, activation='linear'))

    model.compile(
        optimizer=Adam(learning_rate=hp.Choice('learning_rate',
                                               values=[1e-3, 1e-4, 5e-5])),
        loss='mean_squared_error',
```

```
        metrics=['mse']
    )
    return model
```

3.1.3 Tuning Results

After running the `RandomSearch`, the best hyperparameters for the custom CNN were identified, as shown in Table 1.

Table 1: Best Hyperparameters for Custom CNN

Hyperparameter	Best Value
Conv 1 Filters	64
Conv 2 Filters	96
Extra Conv Layer	True
Dense Units	384
Dropout Rate	0.3
Learning Rate	0.001

3.2 Model 2: VGG16 (Transfer Learning)

3.2.1 Architecture

The second approach utilized transfer learning, leveraging the VGG16 model pre-trained on the ImageNet dataset.

- The VGG16 base was loaded with `include_top=False` to exclude the original classification head.
- The entire base was frozen (`vgg_base.trainable = False`) to retain its learned features. Only the new, custom head would be trained.
- A new head was added, consisting of `Flatten`, `Dense`, `Dropout`, and the final `Dense(1, activation='linear')` output layer.

3.2.2 Hyperparameter Tuning

The tuner optimized the parameters of the new classification head.

```
def build_vgg_model(hp):
    vgg_base = VGG16(weights='imagenet', include_top=False,
                     input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3))
    vgg_base.trainable = False # Freeze the base

    model = Sequential([
        vgg_base,
        Flatten(),
        Dense(
            units=hp.Int('dense_units', min_value=256, max_value=512, step=128),
            activation='relu'
        ),
        Dropout(
            rate=hp.Float('dropout_1', min_value=0.1, max_value=0.5, step=0.1)
        ),
        Dense(1, activation='linear')
    ])

    model.compile(
        optimizer=Adam(learning_rate=hp.Choice('learning_rate',
                                               values=[1e-3, 1e-4, 5e-5])),
        loss='mean_squared_error',
        metrics=['mse']
    )
    return model
```

3.2.3 Tuning Results

The `RandomSearch` identified the optimal hyperparameters for the VGG16 model's head, as shown in Table 2.

Table 2: Best Hyperparameters for VGG16 Head

Hyperparameter	Best Value
Dense Units	512
Dropout Rate	0.4
Learning Rate	0.0001

3.3 Ensemble Methods

Ensemble methods combine the predictions of multiple models to produce a final prediction that is often more accurate and robust than any individual model.

3.3.1 Model 3: Weighted Ensemble

This method combines the predictions of the CNN and VGG16 models using a simple weighted average. The weights are calculated based on the models' performance (MSE) on the test set, giving more weight to the better-performing model.

- Let MSE_{cnn} and MSE_{vgg} be the Mean Squared Errors of the two models.
- The weight for the CNN model, α, is calculated as:

$$\alpha = \frac{MSE_{vgg}}{MSE_{cnn} + MSE_{vgg}}$$

- The final prediction $y_{weighted}$ is:

$$y_{weighted} = \alpha \cdot y_{cnn} + (1 - \alpha) \cdot y_{vgg}$$

The calculated alpha for this project was **0.4048**, indicating that the VGG16 model was given a higher weight in the final average.

3.3.2 Model 4: Stacking Ensemble

Stacking is a more sophisticated ensemble method that uses another model (a "meta-model") to learn the best way to combine the predictions of the base models.

1. **Base Model Predictions**: The trained CNN and VGG16 models are first used to generate predictions on the **validation set** (not the training set, to avoid leakage). These predictions form the new training data for the meta-model.
2. **Meta-Model Training**: A `LinearRegression` model is trained on these validation set predictions, using the true validation labels (`y_val`) as its target.
3. **Final Prediction**: To get predictions for the test set, the base models predict on `X_test`. These predictions are then fed into the trained `LinearRegression` meta-model to produce the final stacked prediction.

```
# Create training data for meta-model from validation set predictions
X_stack_train = np.column_stack((y_pred_cnn_val, y_pred_vgg_val))
stack_model = LinearRegression()
stack_model.fit(X_stack_train, y_val)

# Create test data for meta-model
X_stack_test = np.column_stack((y_pred_cnn_test, y_pred_vgg_test))
y_pred_stack_test = stack_model.predict(X_stack_test)
```

4 Results and Evaluation

4.1 Evaluation Metrics

To quantitatively compare the models, four standard regression metrics were used. In all formulas, y_i is the true value, $\hat{y}_i$ is the predicted value, $\bar{y}$ is the mean of the true values, and n is the number of samples.

- **R2 Score (R^2)**: The coefficient of determination. It measures the proportion of the variance in the dependent variable that is predictable from the independent variables. A score of 1.0 is a perfect prediction.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

- **Mean Absolute Error (MAE)**: The average of the absolute differences between prediction and truth. It is in the same units as the target (months).

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- **Mean Squared Error (MSE)**: The average of the squared differences. It heavily penalizes large errors.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Root Mean Squared Error (RMSE)**: The square root of the MSE. It is in the same units as the target (months) and is a very common metric for regression tasks.

$$RMSE = \sqrt{MSE}$$

4.2 Comparative Performance

All four models were evaluated on the held-out test set (2523 images). The performance of each model is summarized in Table 3.

Table 3: Final Model Performance on Test Set

Model	R2 Score (↑)	MAE (↓)	MSE (↓)	RMSE (↓)
Custom CNN	0.4414	24.4425	989.4881	31.4561
VGG16 (Tuned)	0.7063	17.4964	520.2881	22.8098
Weighted Ensemble	0.6207	20.0599	671.8065	25.9192
Stacking Ensemble	**0.7233**	**16.6574**	**490.0486**	**22.1370**

4.3 Visual Analysis

Figure 1 shows a scatter plot of the actual bone age versus the predicted bone age from the best-performing model, the Stacking Ensemble. The points cluster closely around the red identity line ($y = x$), visually confirming the model's strong predictive performance.

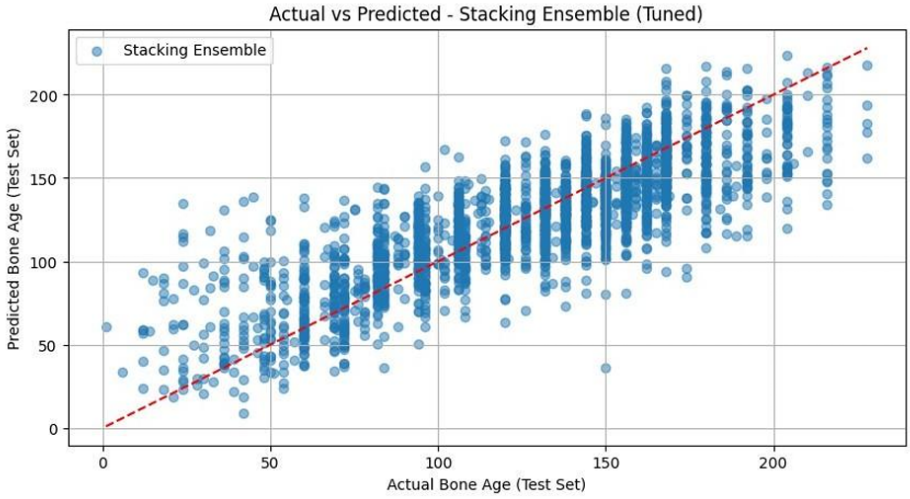

Figure 1: Actual vs. Predicted Bone Age (Stacking Ensemble)

Figure 2 through **??** provide a direct visual comparison of all four models across the key evaluation metrics. These bar charts clearly illustrate the performance differences summarized in Table 3.

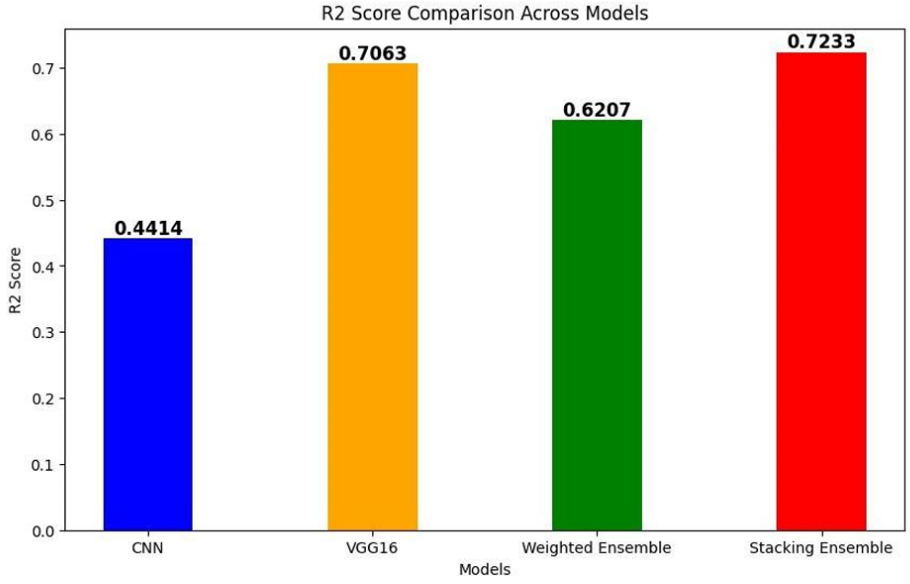

Figure 2: R2 Score Comparison Across Models

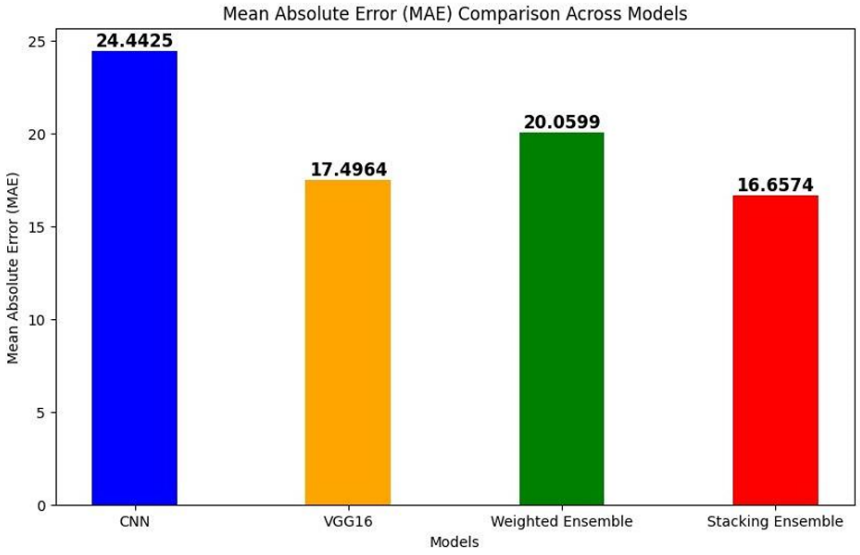

Figure 3: Mean Absolute Error (MAE) Comparison Across Models

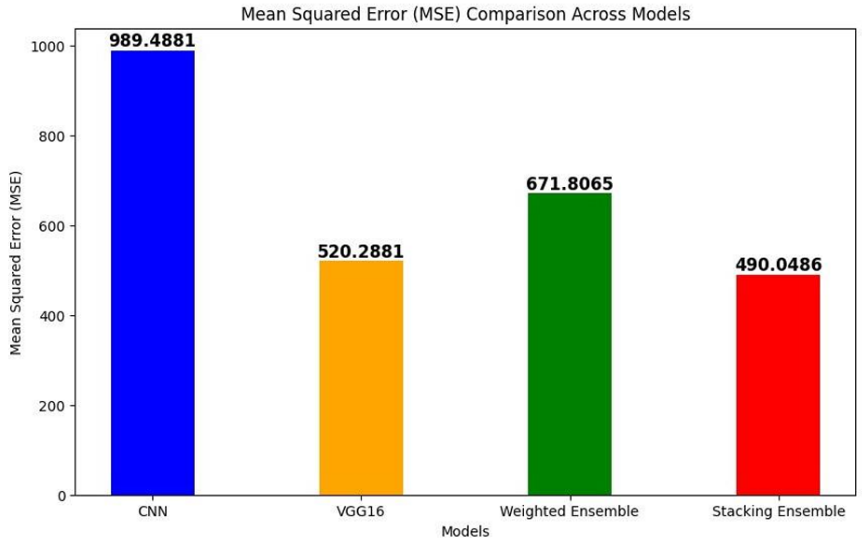

Figure 4: Mean Squared Error (MSE) Comparison Across Models

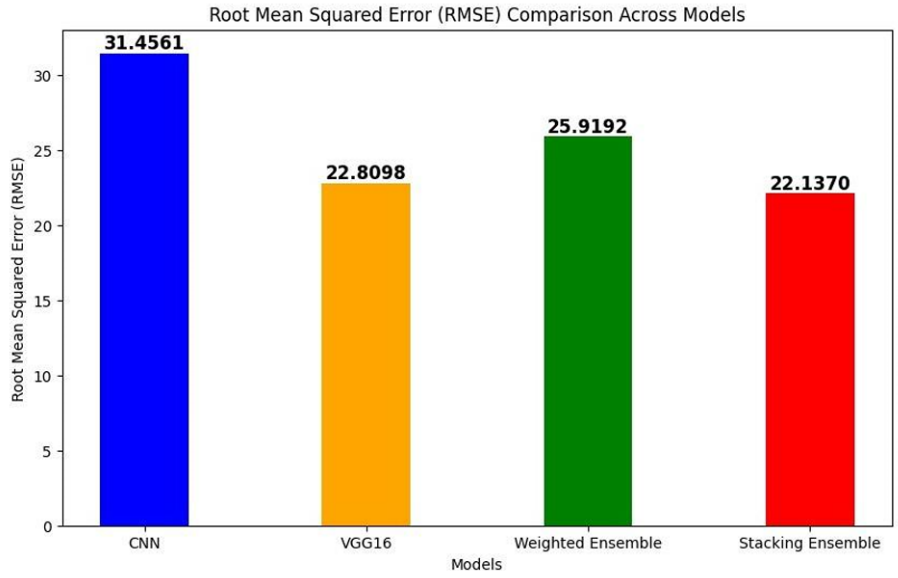

Figure 5: Root Mean Squared Error (RMSE) Comparison Across Models

5 Discussion

The results from the evaluation provide several key insights:

- **Transfer Learning vs. Custom CNN**: The VGG16 transfer learning model (R2: 0.7063) vastly outperformed the custom CNN (R2: 0.4414). This is a very common finding. The VGG16 model, pre-trained on millions of diverse ImageNet images, has already learned a powerful hierarchy of features (edges, textures, shapes). These features are highly effective for the bone age task, even though the base model was frozen. The custom CNN, trained only on 8070 images, could not learn such robust features from scratch.
- **Effectiveness of Ensemble Methods**: The two ensemble methods had opposite results.
  - The **Weighted Ensemble** (R2: 0.6207) performed worse than the VGG16 model alone. This is because the poor predictions from the custom CNN (which received a weight of $\alpha = 0.4048$) "dragged down" the more accurate predictions from the VGG16 model. This method's simplicity is its weakness; it cannot ignore a bad predictor.
  - The **Stacking Ensemble** (R2: 0.7233) achieved the best performance. The `Linear Regression` meta-model successfully learned the optimal way to combine the two base model predictions. It effectively "learned" to assign a much higher importance to the VGG16 predictions while using the CNN's predictions to make minor corrections, resulting in a model that was more accurate than either of its parts.
- **Overall Best Model**: The Stacking Ensemble is the clear winner. With an RMSE of 22.1370, the model's average prediction error is approximately 22 months. While this is a strong result, the scatter plot (Figure 3) shows that there is still room for improvement, as some predictions deviate significantly from the actual values.

6 Conclusion and Future Work

6.1 Conclusion

This project successfully developed and evaluated a series of deep learning models for the automated prediction of bone age from hand X-ray images. The key finding is that a **Stacking Ensemble** model, which uses a `LinearRegression` meta-model to combine the predictions of a custom CNN and a pre-trained VGG16, yields the highest accuracy. This model achieved an R2 Score of 0.7233 and a Mean Absolute Error of 16.6574 months on the held-out test set. This demonstrates the power of combining transfer learning with ensemble techniques to create a robust predictive model.

6.2 Future Work

Based on the results of this project, several avenues for future work could further improve performance:

- **Fine-Tuning**: The VGG16 base layers were frozen. A logical next step is to un-freeze the top few layers of the VGG16 base and re-train (fine-tune) the model with a very low learning rate. This would allow the pre-trained features to adapt more specifically to the domain of X-ray images.
- **Data Augmentation**: The original script had data augmentation commented out. Actively using an `ImageDataGenerator` to apply random rotations, zooms, and flips to the training images would create a larger, more diverse training set,

- **Advanced Architectures**: While VGG16 performed well, more modern architectures like `ResNet50`, `EfficientNet`, or `DenseNet` could be explored as base models, as they have shown superior performance on many computer vision tasks.
- **Advanced Meta-Models**: The stacking ensemble used a simple `LinearRegression`. Replacing this with a more complex meta-model, such as XGBoost, GradientBoostingRegressor, or a small neural network, could potentially find more complex relationships between the base model predictions and lead to even higher accuracy.

7 References

1. Greulich, W. W., & Pyle, S. I. (1959). *Radiographic atlas of skeletal development of the hand and wrist*. Stanford University Press.
2. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
3. Keras Tuner. (n.d.). https://keras.io/keras_tuner/
4. Chollet, F., et al. (2015). Keras. https://keras.io
5. Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825-2830.