Group 4

Project Report

Predicting hearing threshold using Brain MRI scans

Charanjit Singh (C15246652)

Parampreet Singh (C19377466)

Teajsahree Challagundla (C12214588)

Table of Contents

Abstract

Hearing loss significantly affects the quality of life, particularly among older adults. Advances in medical imaging and machine learning offer novel approaches for early detection and intervention. This project employs brain MRI scans to predict hearing thresholds, utilizing machine learning models such as Convolutional Neural Networks (CNNs), Support Vector Regression (SVR), and eXtreme Gradient Boosting (XGBoost).

The dataset consists of 171 subjects' gray matter images, which were processed and augmented to enhance the training of the models. The CNN model, designed specifically for 3D data, includes multiple convolutional, max pooling, and dropout layers to manage overfitting and effectively capture spatial hierarchies. Comparative performance assessments with SVR and XGBoost highlighted CNN's superior performance in minimizing mean absolute error (MAE) and mean squared error (MSE).

These findings suggest that machine learning techniques, particularly CNN, are promising tools for analyzing brain MRI data to predict hearing thresholds. This approach demonstrates the potential of integrating advanced analytical methods into healthcare diagnostics, potentially paving the way for innovative preventive measures in auditory health management.

Introduction

Hearing loss, or hearing impairment, involves a reduction in the ability to perceive sounds and can range from mild hearing difficulties to complete deafness. This condition can affect one or both ears and is influenced by a variety of factors. It is categorized into three primary types:

Conducive Hearing Loss

Conductive hearing loss occurs when sound waves are obstructed in the outer or middle ear. Common causes include ear infections that lead to fluid build-up and blockages, excessive earwax that blocks the ear canal, otosclerosis which causes a tiny bone in the middle ear to stiffen and lose its ability to vibrate, and perforated eardrums from injuries or infections that disrupt sound transmission.

Sensorineural Hearing Loss

Sensorineural hearing loss, the most common type, is especially linked to aging and chronic noise exposure. It results from damage to the sensory or hair cells in the cochlea of the inner ear, or to the nerve pathways from the inner ear to the brain. The cochlea's hair cells are delicate structures that translate sound vibrations into electrical signals sent to the brain. As people age, these hair cells gradually deteriorate and lose their function, a process known as presbycusis. This type of hearing loss can also be exacerbated by prolonged exposure to loud noises, which can irreversibly damage these cells, and by ototoxic drugs and certain diseases like Meniere’s disease that negatively affect the inner ear.

Mixed Hearing Loss

Mixed hearing loss encompasses elements of both conductive and sensorineural hearing loss, where mechanical and sensory impairments coexist. Moreover, hearing loss can also be characterized by its onset—progressive hearing loss develops slowly over time and is often due to aging or noise exposure, while sudden hearing loss can occur almost instantaneously, often due to an underlying medical condition, trauma, or exposure to certain drugs.

Treatment options for hearing loss vary widely, depending primarily on the underlying cause of the impairment. Conductive hearing loss can often be managed with medical and surgical interventions, while sensorineural hearing loss may require the use of hearing aids or cochlear implants. In addition to these targeted treatments, preventative measures play a crucial role in maintaining auditory health. Managing noise exposure and avoiding ototoxic medications are essential strategies for preventing damage to the cochlea's hair cells, which helps in preserving hearing capabilities.

Despite these precautions, hearing loss becomes more prevalent with age. The auditory system's metabolic and sensory components are particularly susceptible to age-related decline. A report from the National Institutes of Health (NIH) highlights that over half of Americans aged 75 and older experience hearing difficulties (*figure 1*). This statistic not only emphasizes the widespread nature of the condition but also underscores the importance of early intervention to address and understand the factors contributing to hearing impairment so we can mitigate the impacts of this pervasive health issue.

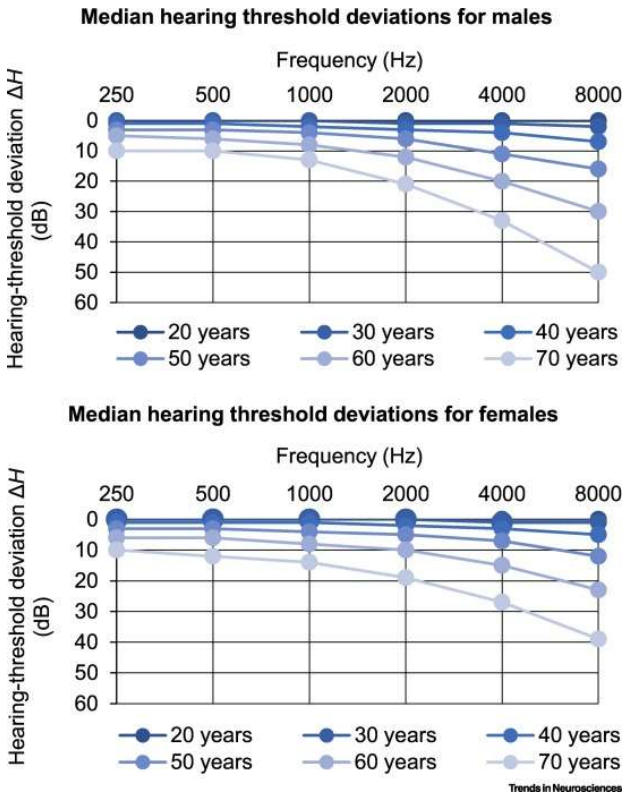


Figure 1: Graphic of High-Frequency Threshold Elevation as a Function of Age and Gender On a Pure-Tone Audiogram.

Magnetic Resonance Imaging

In pursuit of such understanding, brain Magnetic Resonance Imaging (MRI) emerges as a crucial tool. It is a sophisticated imaging technique that plays a pivotal role in the medical field, especially in the diagnosis and monitoring of various neurological conditions. Utilizing strong magnetic fields and radio waves, MRI provides exceptionally detailed images of the brain's structures, making it an invaluable tool in understanding complex brain functions and pathologies.

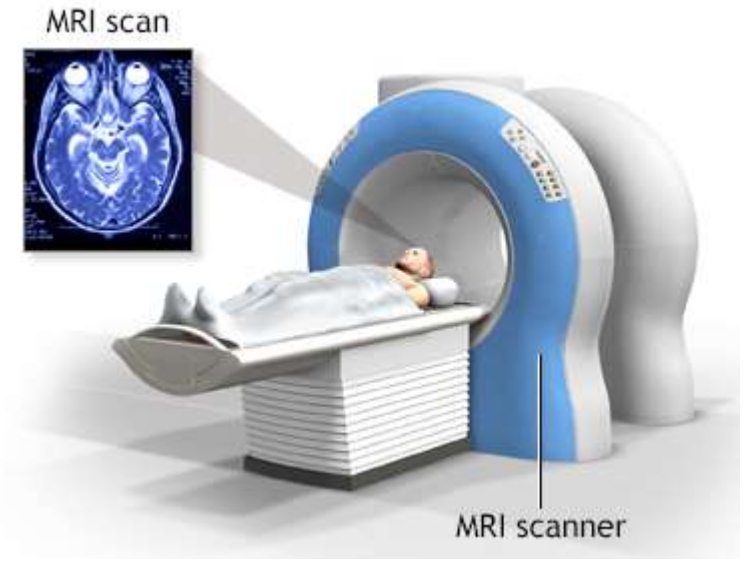


Figure 2: MRI Scanner

MRI has been instrumental in predicting and diagnosing several neurological diseases well before clinical symptoms manifest. For example, in the case of Alzheimer's disease, MRI can detect early signs of brain atrophy, particularly in the hippocampus and other regions vulnerable to neurodegenerative processes, allowing for earlier intervention (*figure 3*). Similarly, in multiple sclerosis, MRI is used to identify and monitor the appearance of lesions in the central nervous system, even when symptoms are not apparent, which helps in managing the disease proactively.

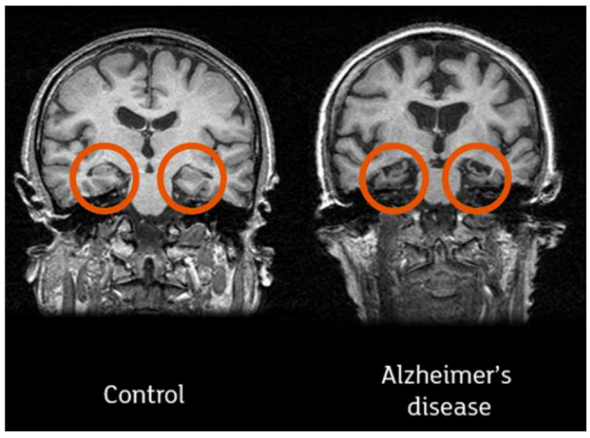


Figure 3: Using Brain MRI Scans to detect Alzheimer's Disease.

The application of MRI extends to the study of sensorineural hearing loss. This type of hearing loss, which results from damage to the hair cells in the cochlea or the auditory neural pathways, can also benefit from the detailed imaging that MRI provides. By capturing the structural changes in the brain regions associated with auditory processing, MRI offers a window into the biological basis of hearing loss. This capability allows researchers to correlate specific patterns of brain morphology with auditory function. For instance, changes in the volume of the auditory cortex or alterations in the connectivity of neural networks involved in sound processing can be detected, providing insights into the extent of hearing impairment and guiding treatment strategies.

In summary, brain MRI emerges as a crucial tool in the pursuit of understanding sensorineural hearing loss, much like its use in predicting other neurological diseases. It provides detailed images that reveal both structural and functional changes in the brain, offering a comprehensive approach to tackling this prevalent health issue.

MRI Scan Results

Magnetic Resonance Imaging (MRI) produces detailed images of the body's internal structures, particularly soft tissues, which are captured in the form of raw data by the MRI scanner. This raw data is then processed and converted into a format that can be easily analyzed and shared. One of the most common formats for storing MRI data is the NIfTI-1 format, known as .nii files. These files are adept at handling three-dimensional data, encapsulating not only the image itself but also an extensive header that contains important metadata about the scan. This metadata includes details such as the scan dimensions, voxel size, data type, and orientation of the image. In the case of three-dimensional data, the .nii file organizes the data into a 3D matrix where each element, or voxel, represents a tiny cube of the scanned tissue. The intensity value of each voxel corresponds to a specific property of the tissue, such as its density or water content, providing rich spatial information about the internal structures. This format is highly advantageous for medical analysis and research, as it supports extensive manipulation and visualization of the 3D data, facilitating a deeper understanding of complex anatomical and pathological conditions.

Problem Statement

Traditional methods of diagnosing hearing loss are predominantly reactive rather than proactive. There is a critical need for innovative approaches that can identify the onset of hearing impairment earlier and more accurately.

Our project leverages brain MRI scans for detailed visualization of brain structures, to address this need. This project explores the potential of these scans, combined with advanced data mining techniques, to predict hearing thresholds. By doing so, we aim to develop a predictive model that can serve as a non-invasive method to identify individuals at risk of hearing loss.

The integration of MRI data with machine learning models presents a novel approach to understanding the intricate relationship between brain morphology and auditory function. Our goal is to harness these technologies to not only predict a person’s hearing ability but also facilitate earlier interventions. This could significantly contribute to the efforts in auditory health preservation, offering a proactive solution to the silent epidemic of hearing loss.

Data Preparation

The dataset is a collection of brain MRI scans from 171 subjects, specifically focusing on gray matter images. Gray matter images highlight the regions of the brain composed primarily of neuronal cell bodies, dendrites, and unmyelinated axons, which are crucial for processing information and executing brain functions. These images are stored in the .nii file format, which is standard for medical imaging as it supports three-dimensional data along with associated metadata. This comprehensive format is instrumental in our analysis as it captures the complete anatomy of the brain required for detailed examination.

We are also given a csv file which contains the hearing thresholds corresponding to each of the 171 scans at the pure tone frequencies of 500 and 4000. The given label data is shown below:

File Name	PT500	PT4000
smwp1_0001	25	73
smwp1_0002	5	10
smwp1_0003	5	45
smwp1_0004	5	15
smwp1_0005	3	63
smwp1_0006	10	18
smwp1_0007	10	38
smwp1_0008	18	45
smwp1_0009	8	83
smwp1_0010	15	53
smwp1_0011	23	38

Data Visualization

To visualize .nii files effectively, Python, with libraries such as NiBabel for reading .nii files and Matplotlib for plotting, provides a straightforward solution. The Appendix has an example code that demonstrates how to load a .nii file using NiBabel, extract its data as a NumPy array, and then visualize a slice of the MRI scan. The images of extracted MRI scans can be seen below:

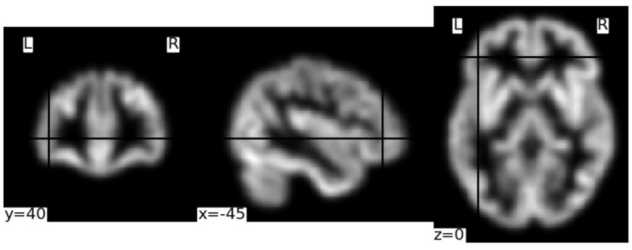


Figure 4: Brain MRI Scans obtained from .nii files.

The script begins by importing the necessary libraries, then loads a .nii file using NiBabel. The image data is extracted into a NumPy array where its shape corresponds to the dimensions of the brain scan in three axes (X, Y, Z). The show_slice function is a utility to visualize a single slice. The code selects the middle slice from each of the three dimensions to provide a comprehensive view of the brain's structure in the axial, coronal, and sagittal planes. The slices are displayed using Matplotlib, which plots them in grayscale. This visualization technique is particularly useful for examining the spatial structure of gray matter in the brain, helping in analyses related to anatomical features and potential abnormalities.

Data Preprocessing

The raw MRI scans must undergo several preprocessing steps before they can be used for machine learning. These steps ensure that the data fed into our models is clean, uniform, and structured, maximizing the efficacy of the algorithm's learning process.

In our study involving a dataset of 171 brain MRI scans stored in .nii file format, the first step in the data preprocessing involved extracting these scans into NumPy arrays. These numpy arrays were then separated into test and train matrices. Each array in test and train matrix represents the 3D structure of the brain, from which a specific slice was selected. This slice was chosen because it encompasses the region containing the auditory components of the brain. By focusing on this targeted slice, we effectively eliminated irrelevant sections of the image that do not contribute to our specific study of hearing loss. This reduction not only streamlines the computational load but also sharpens the focus of the analysis, enhancing the precision of our predictive model.

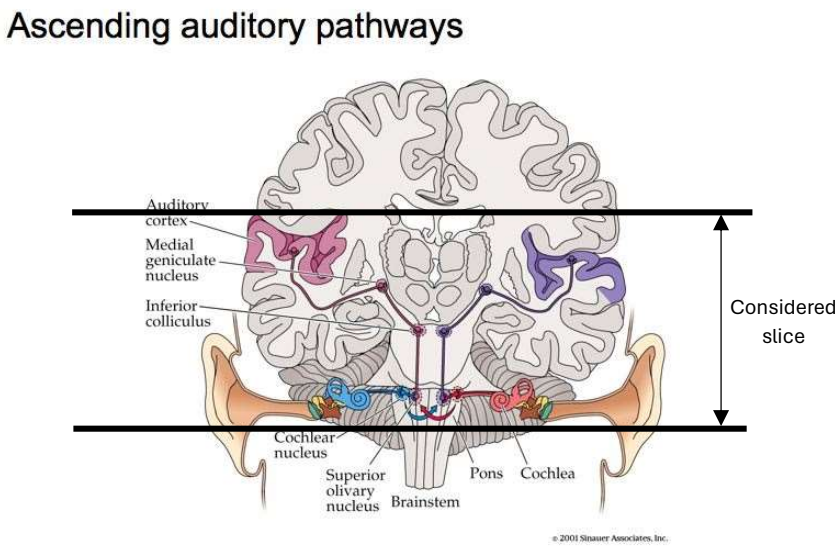


Figure 5: The slice MRI scan slice considered for training neural network.

Following the extraction and selection of the relevant slices, the next crucial step was data normalization. Normalizing the test and train image data involves adjusting the intensity values so that they fall within a standard range, usually between 0 and 1. This step is essential because it

standardizes the data, making it easier for machine learning algorithms to process. Normalization helps in mitigating the effects of contrast variations between different scans, which can arise due to differences in machine calibration, scanning protocols, or patient movement. By normalizing the data, we ensure that the model's performance is influenced by meaningful anatomical differences rather than by these technical discrepancies.

Finally, to augment the dataset and enhance the robustness of our predictive model, the processed slices of training data were concatenated with themselves, effectively doubling the number of training samples. This technique, known as data augmentation, is particularly useful in scenarios where the dataset is relatively small—as is the case with our 171 scans—since it helps prevent overfitting and improves the generalizability of the model. By training on these augmented data, the model can learn more comprehensive and diverse patterns, leading to better performance in predicting hearing thresholds from MRI scans.

Machine Learning Approaches

In the pursuit of predicting hearing thresholds from brain MRI scans, we explored three advanced machine learning models, each with its own strengths and weaknesses in terms of performance, computational demands, and suitability for the task at hand.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are suited for predicting hearing thresholds from brain MRI scans due to their sophisticated ability to process and analyze image data. As advanced deep learning models designed for grid-like data structures, such as those found in MRI images, CNNs are highly efficient at extracting and interpreting spatial hierarchies of features directly from the scans. This ability is crucial in medical imaging, where identifying subtle spatial relationships and patterns within the brain's anatomy can directly influence diagnostic accuracy. The architecture of CNNs enables them to excel in image-related tasks by effectively capturing the intricate details and variations in brain structures associated with hearing loss.

Moreover, CNNs can automate the feature extraction process, which traditionally requires extensive manual intervention and expert knowledge, thereby reducing the time and potential bias involved in feature selection. This makes CNNs particularly valuable for interpreting complex MRI data, where distinguishing between relevant and irrelevant features can significantly impact the outcomes of predictive models. However, the use of CNNs in such applications does come with challenges, such as high computational costs and a tendency to overfit. These can be mitigated by adopting strategies like data augmentation, regularization techniques, and employing appropriate hardware to handle the processing demands of high-resolution 3D MRI scans. By leveraging their image-processing capabilities, CNNs hold the potential to enhance the precision and effectiveness of models aimed at predicting hearing thresholds, ultimately aiding in the early detection and treatment of hearing impairment.

Model Architecture

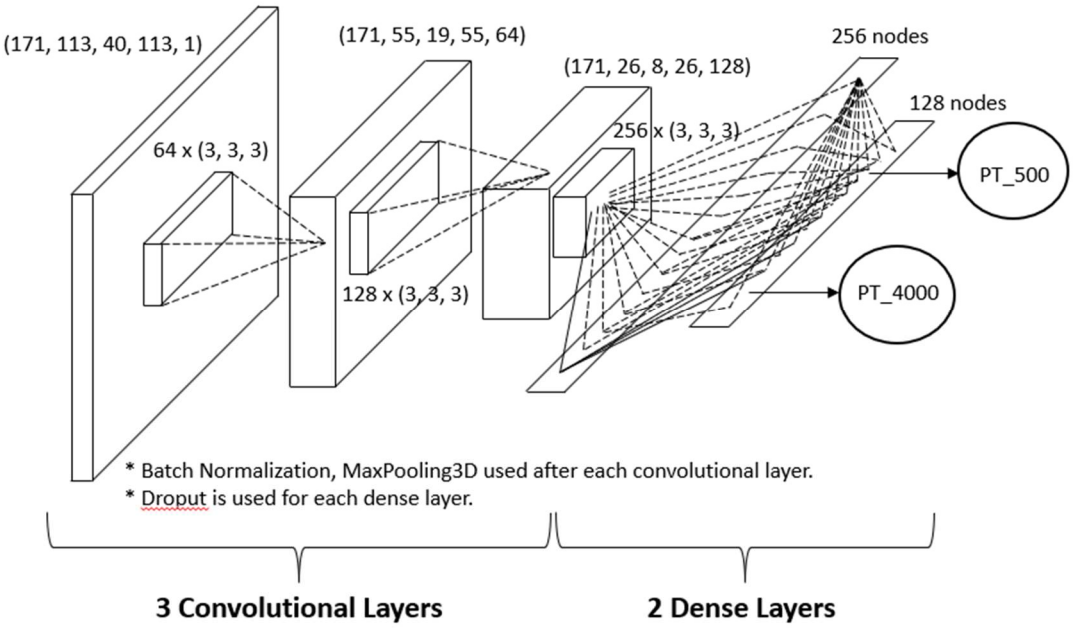


Figure 6: CNN architecture used for threshold predictions.

The model architecture finalized for Convolutional Neural Network (CNN) is tailored for image analysis, featuring a sequence of layers designed to process and learn from three-dimensional data. Initially, the input layer receives images with dimensions (171, 113, 40, 113, 1), indicative of a single-channel 3D image, such as a specific MRI slice.

This CNN consists of three convolutional layers, each followed by batch normalization and 3D max pooling operations. The first convolutional layer applies 64 filters of size (3, 3, 3), effectively capturing local patterns within the input data. The second layer deepens the network with 128 filters of the same size, further refining the feature maps generated by the previous layer. The third convolutional layer expands this with 256 filters, allowing the network to detect even more complex features within the images.

After feature extraction, the network transitions to two dense layers, which are fully connected neural layers that interpret the features extracted by the convolutional layers. These dense layers consist of 256 and 128 nodes respectively, leading to the final prediction layers named PT_500 and PT_4000, suggesting the model predicts thresholds at 500 Hz and 4000 Hz.

Batch normalization is used after each convolutional layer to standardize inputs and stabilize learning, while dropout is implemented in the dense layers to prevent overfitting by randomly setting a fraction of input units to 0 during training. This architecture is designed to effectively learn from 3D data, with each layer contributing to a hierarchy of feature detection and abstraction, ultimately aimed at predicting auditory thresholds from brain MRI scans with high precision.

Principle Component Analysis & Support Vector Regression (SVR)

The PCA-SVR approach embodies a two-stage process aimed at enhancing the predictive modeling of complex datasets, such as those derived from brain MRI scans. The initial phase of this strategy involves the application of Principal Component Analysis (PCA), a dimensionality reduction technique that transforms the high-dimensional data into a set of linearly uncorrelated variables known as principal components. This is particularly beneficial for MRI data, which often contains a large number of voxels that may not all contribute meaningful information for regression tasks. By reducing the dimensionality of the data to the most informative components (10 components in our case), PCA serves to concentrate the variance into fewer dimensions, thus simplifying the dataset while retaining the most critical features.

Following dimensionality reduction, the Support Vector Regression (SVR) model is trained on this more tractable representation of the data. SVR operates by attempting to find a hyperplane that best fits the data, with C (1) and epsilon (0.2) parameters governing the regularization and the margin of tolerance for errors, respectively. This machine learning algorithm is well-suited for regression, capable of modeling both linear and non-linear relationships through the selection of appropriate kernel functions.

For the application of this approach, we constructed a pipeline that chains PCA and SVR together for each target variable in a multivariate regression context. The loop iterates over each image matrix which is flattened, applying PCA to reduce dimensionality and then fitting an SVR model to the lower-dimensional data. The result is a model, trained to predict the hearing thresholds from the processed MRI data. This PCA-SVR approach is adept at handling the complexity and size of the MRI data, enhancing the model's efficiency and potentially boosting its predictive accuracy.

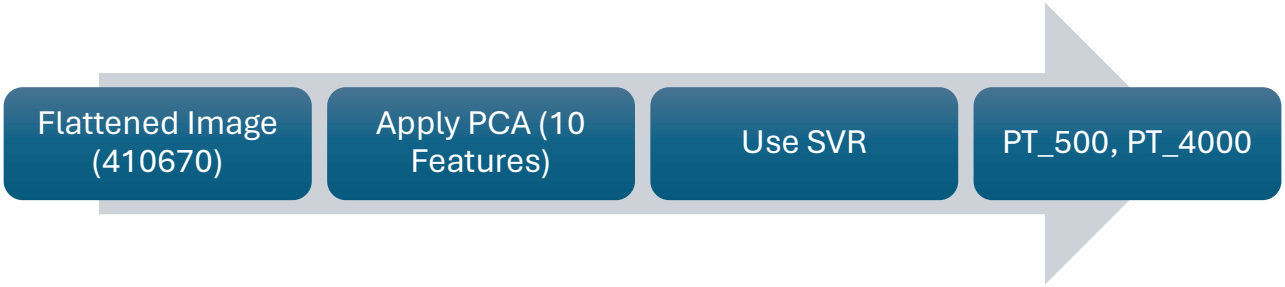


Figure 7: Using PCA+SVR for threshold predictions.

XGBoost

In the realm of machine learning and data analysis for medical imaging, particularly with complex datasets like brain MRI scans, feature selection becomes a crucial step. The provided code snippet outlines a feature selection strategy followed by a predictive modeling phase using XGBoost, an advanced gradient boosting framework.

The feature selection phase begins by employing the SelectKBest method with an f_regression scoring function to isolate the top k features from the training data that exhibit the most significant relationship with each target variable. By iterating over each target, a combined mask is created that encompasses the best features across all targets, thereby refining the feature space to only the

most relevant attributes. This process not only reduces the dimensionality of the data, making it more manageable for computational processes, but also helps in removing noise and potentially irrelevant information, which could lead to overfitting or obfuscate true predictive patterns.

Once the most pertinent features are selected and applied to both the training and test sets, the XGBoost model comes into play. With a MultiOutputRegressor wrapper, the XGBoost model is adapted to handle multiple regression outputs—a necessary feature for predicting various hearing thresholds from MRI data. To optimize the model, a grid search across a pre-defined hyperparameter space is conducted, utilizing cross-validation to ensure robustness and prevent overfitting. This meticulous search for the best parameters underlines the careful balance required to maintain model complexity and predictive power.

The final model is trained with early stopping to enhance efficiency, halting the training process if the validation metric does not improve for a set number of rounds. The training and validation Mean Squared Error (MSE) are monitored over epochs, and their progression is graphically represented, providing an insightful visual of the model's learning curve. The convergence of these two lines indicates a well-fitted model, while divergence could signal overfitting to the training data.

This sophisticated approach—combining feature selection and the powerful XGBoost algorithm with its gradient boosting capabilities—illustrates a methodical and analytical pathway designed to distill complex MRI data into accurate predictions of hearing thresholds. This methodology exemplifies the confluence of statistical techniques and machine learning prowess in tackling high-dimensional data to glean valuable predictions in medical science.

Models Comparison

In our study, we employed Mean Squared Error (MSE) and Mean Absolute Error (MAE) as metrics to analyze and compare the performance of our regression models. MSE is calculated by averaging the squares of the differences between the predicted and actual values (*Equation 2*), giving us a comprehensive measure of the model's prediction accuracy while heavily penalizing larger errors. This makes MSE particularly sensitive to outliers in the predictions. On the other hand, MAE measures the average magnitude of the errors by averaging the absolute differences between predicted and actual values (*Equation 1*), providing a more straightforward and robust indication of the average prediction error. By using both MSE and MAE, we were able to gain a dual perspective on our models' performance: MSE offered insight into the overall prediction accuracy with a focus on large errors, whereas MAE presented a clear and easily interpretable metric that is less affected by extreme deviations. This multifaceted evaluation approach allowed us to thoroughly assess and compare the effectiveness of our regression models in predicting hearing thresholds from brain MRI scans, ensuring a balanced and comprehensive analysis.

$$MAE = \frac{1}{N}\sum_{i=1}^{n}(|y_{i1} - \widehat{y_{i1}}|+|y_{i2} - \widehat{y_{i2}}|)$$

Equation 1: Calculation of Mean Absolute Error

$$MSE = \frac{1}{N}\sum_{i=1}^{n}((y_{i1} - \widehat{y_{i1}})^2 + (y_{i2} - \widehat{y_{i2}})^2)$$

Equation 2: Calculation of Mean Squared Error

CNN

CNN has been successful in identifying and extracting relevant features from the MRI images, which is crucial for the prediction task at hand. The effectiveness of the feature extraction by the CNN layers is evident, indicating that the model has been learning the distinguishing features of the MRI scans that correlate with hearing threshold levels.

Two metrics are used to evaluate the model:

MAE	MSE
11.14	221.04

Analysis of Loss Curves

The loss curves provide insight into the model's learning over time, depicted over the number of epochs (each epoch represents a complete pass through the full training dataset).

The training loss curves of our models shed light on the significant role that regularization and dropout techniques play in curtailing overfitting. Initially, our model exhibited overfitting, as indicated by the first graph, where the training loss decreased sharply while the validation loss remained relatively high. Overfitting is a common problem where a model learns the training data so closely, including its noise, that its ability to generalize to unseen data is compromised. Regularization addresses this by adding a penalty for complexity, discouraging the model from fitting excessively to the training data.

To enhance the model's generalization further, dropout is utilized as an additional form of regularization. Dropout works by randomly deactivating a subset of neurons during each training iteration, which prevents the network from becoming overly reliant on any specific set of neurons. This technique encourages the development of a more robust network with better generalization properties, as it simulates the training of a large number of neural networks with varying architectures.

Upon analyzing the loss curves from the training of our models, we observe distinct behaviors indicative of their learning dynamics and generalization capabilities. The first graph (*figure 8*) demonstrates a scenario of low regularization, where the training loss decreases significantly, yet the validation loss remains high, signaling a clear case of overfitting. The model learns the training data too well, including its noise and outliers, which does not generalize well to the unseen validation data.

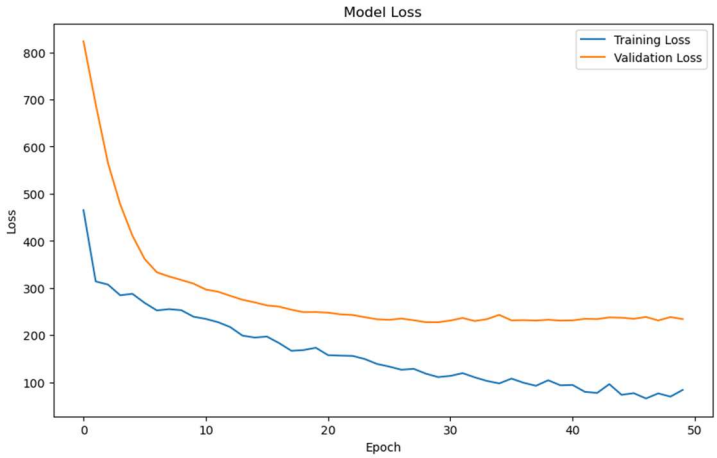


Figure 8: Model training with low regularization (lambda = 0.001)

The second graph (*figure 9*) illustrates the effect of implementing both higher regularization rates and dropout. We observe the training and validation losses converging, implying that the model is now learning general patterns that perform well on both the training and validation datasets. This synergy between regularization and dropout fosters a model that is less prone to overfit and more stable when exposed to new data.

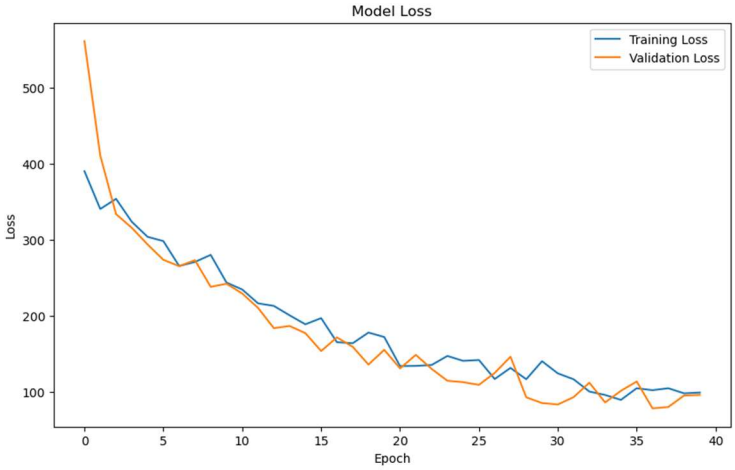


Figure 9: Model training with increased regularization (lambda = 0.01)

The third graph (*figure 10*) depicts the model with the lowest validation loss, showcases the benefits of these techniques in harmony. The consistent and parallel trends of the loss curves across training and validation phases demonstrate a model that is effectively regularized and has a strong predictive performance. By carefully calibrating the dropout rate and the regularization parameters, we fine-tune our model to achieve an optimal balance, ensuring it is neither underfit nor overfit. This

balanced model stands as a robust predictor of hearing thresholds from brain MRI scans, capable of delivering reliable and accurate results.

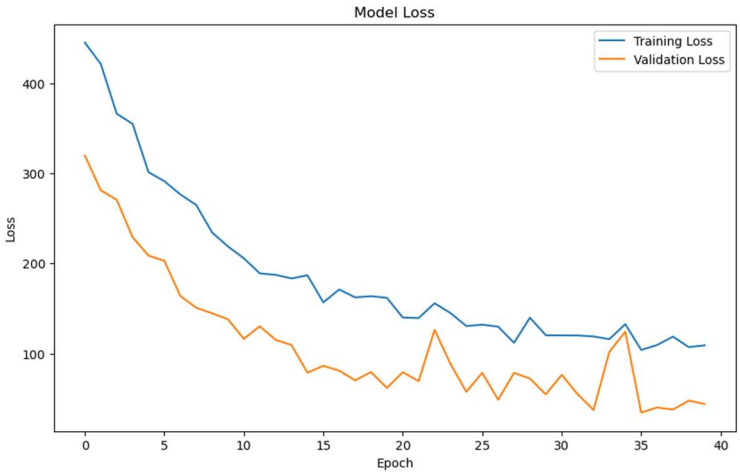


Figure 10: Model training which results in the lowest validation loss.

PCA + SVR

The performance of the PCA+SVR model on our dataset, as quantified by the Mean Absolute Error (MAE) and Mean Squared Error (MSE), was very close, but less than the CNN. This outcome suggests that the PCA+SVR model, despite its sophisticated approach to dimensionality reduction and regression, might not have captured the underlying patterns in the data as effectively as the CNN. A probable reason for this could be the inherent trade-off that comes with dimensionality reduction using PCA. While PCA is effective in reducing noise and computational complexity by projecting the data onto a lower-dimensional space, it may also inadvertently discard subtle yet crucial information that is necessary for accurate prediction of hearing thresholds. The SVR component of the model relies on the reduced feature set provided by PC, which is expected to preserve the important information from the large dataset into a more compact feature set. Consequently, this could lead to a less accurate model, reflected in higher MAE and MSE values.

MAE	MSE
12.45	244.48

XGBoost

XGBoost's inherent ability to handle incomplete data has allowed for uninterrupted training, underscoring its robustness in dealing with real-world, imperfect datasets. Its built-in features to combat overfitting have proven effective, as observed through stable validation loss, suggesting that the model has successfully generalized beyond the training data. This balance between learning from the training set and maintaining predictive power on unseen data is crucial for the deployment of machine learning models in practical settings.

MAE	MSE
14.38	338.40

In terms of computational efficiency, XGBoost's efficient implementation of the gradient boosting framework has ensured scalability and swift processing, effectively managing the voluminous data derived from brain MRI scans. But the model's evaluation using Mean Absolute Error (MAE) and Mean Squared Error (MSE) has yielded an MAE of 14.38 and an MSE of 338.4. These metrics highlight the cost of large errors more prominently, which can be due to the fact the XGBoost is not able to capture the relevant features as well as CNN or SVR+PCA.

The analysis of the training and validation loss curves (*figure 11*) has provided additional insights into the model's learning dynamics. The decline in training MSE is indicative of the model effectively capturing the patterns within the dataset, while the plateauing of the validation MSE suggests that the model's capacity to learn has been maximized given the current hyperparameter settings. The validation curve does not show much improvement with epochs. Overall, The XGBoost algorithm performed the worst between all the algorithms.

Optimal performance was attained through meticulous hyperparameter tuning. The parameters that contributed to this finely tuned model include a **colsample_bytree of 0.7**, ensuring diversity in the features that the individual trees consider, and a **learning_rate of 0.2**, which allowed for rapid convergence to a high-performing model without overshooting the optimum. The **max_depth** was set to **4** to prevent the model from becoming overly complex, and **n_estimators** was chosen to be **200**, providing a substantial yet computationally manageable number of trees in the ensemble. Lastly, a **subsample rate of 0.9** encouraged model robustness, further aiding in the reduction of overfitting.

Figure 11: MSE observed during training using XGBoost.

Conclusion

Model	Evaluation Metric	Result
CNN	MAE	**11.14**
	MSE	**221.04**
SVR	MAE	**12.45**
	MSE	**244.48**
XGBoost	MAE	**14.38**
	MSE	**338.4**

In conclusion, our exploration into the predictive modeling of pure tone thresholds from brain MRI scans has been insightful and revealing. Among the machine learning models evaluated, the Convolutional Neural Network (CNN) emerged as the most proficient, reflecting its robust capability to handle the intricate spatial hierarchies and complex patterns inherent in MRI data. The CNN's superior performance can be attributed to its deep architecture that excels in image-based tasks, making it particularly suitable for our high-dimensional dataset. The predictions obtained from CNN are given in the table below.

PT_500 (Actual)	PT_500 (Predicted)	PT_4000 (Actual)	PT_4000 (Predicted)
10	**14.4**	**13**	**7.70**
5	**6.7**	**0**	**2.82**
65	**63.91**	**28**	**24.41**
5	**4.67**	**3**	**2.71**
30	**28.21**	**15**	**12.43**

Following the CNN, the PCA+SVR model demonstrated the next level of performance. While this approach benefited from dimensionality reduction and the ability to handle non-linear patterns, it appears that some crucial information pertinent to auditory processing might have been lost in the PCA stage, impacting the SVR's ability to make precise predictions.

PT_500 (Actual)	PT_500 (Predicted)	PT_4000 (Actual)	PT_4000 (Predicted)
40	36.37	18	12
20	21.91	5	5.56
18	26.98	25	11.1
25	20.91	8	4.49
15	37.22	5	8.5

Finally, XGBoost, ranked the last in terms of performance. It had the highest error in prediction, as reflected by MAE and MSE values. The least performance of XGBoost on an MRI dataset, particularly when compared to models like CNNs or PCA+SVR, could be due to the fact that it cannot automatically extract complex features from raw MRI images, which may lead to less informative features being used for training the model. Additionally, XGBoost has a vast hyperparameter space, and finding the optimal set can be challenging. Inadequate hyperparameter tuning can lead to underfitting or overfitting, which can degrade the model's performance. While XGBoost is efficient with tabular data, the computational resources required for processing 3D MRI data might limit the extent to which XGBoost can be tuned and trained, potentially affecting its performance.

PT_500 (Actual)	PT_500 (Predicted)	PT_4000 (Actual)	PT_4000 (Predicted)
15	15.39	10	10.62
40	43.12	18	12.05
25	22.61	13	15.73
10	15.26	3	2.34
13	11.70	13	9.15

Overall, the models' performances have provided us with valuable insights into the strengths and limitations of different machine learning strategies for medical image analysis. These findings will not only inform future research directions but also underscore the importance of model selection tailored to the specific features of the dataset in question. As we continue to refine these models and explore new methodologies, we move closer to the goal of non-invasive, accurate prediction of hearing impairment, a development with significant implications for improving the quality of life for those affected by hearing loss.

Appendix

CPSC 8650 - DATA MINING PROJECT

Predicting Hearing Thresholds from Brain MRI Scans

GROUP 4

MEMBERS: Charanjit Singh, Parampreet Singh, Tejashree Challagundla

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

In [ ]:
```
#LIBRARIES
import pandas as pd
import nibabel as nib
import os
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split, GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv3D, MaxPooling3D, Flatten, Dense, Dropout,
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.svm import SVR
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_absolute_error, mean_squared_error
from tensorflow.keras.regularizers import l2
import xgboost as xgb
from sklearn.multioutput import MultiOutputRegressor
from sklearn.feature_selection import SelectKBest, f_regression
from xgboost import XGBRegressor
```

In [ ]:
```
len(tf.config.list_physical_devices('GPU'))
```

Out[ ]:
```
1
```

Data Preparation:

- Process the MRI gray matter images to a consistent format suitable for deep learning models.
- Extract and prepare the hearing threshold data for the prediction task.

In [ ]:
```
# Load the dataset from the provided Excel file
labels_path = "C:\\Users\\chara\\OneDrive\\Desktop\\DM\\threshold_labels.csv"

# Load the dataset from the Excel file
threshold_labels = pd.read_csv(labels_path)  # This assumes the required data is ir

# Display the first few rows of the dataset to understand its structure
print(threshold_labels.head(10))
```

	ID	PT500	PT4000
0	smwp1_0001	25	73
1	smwp1_0002	5	10
2	smwp1_0003	5	45
3	smwp1_0004	5	15
4	smwp1_0005	3	63
5	smwp1_0006	10	18
6	smwp1_0007	10	38
7	smwp1_0008	18	45
8	smwp1_0009	8	83
9	smwp1_0010	15	53

In [ ]:
```
threshold_labels.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 171 entries, 0 to 170
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   ID      171 non-null    object
 1   PT500   171 non-null    int64 
 2   PT4000  171 non-null    int64 
dtypes: int64(2), object(1)
memory usage: 4.1+ KB
```

Processing MRI Scans

For the MRI scans, we'll need to:

- **Standardize Image Size:** Ensure all images are of the same dimensionality for model input.
- **Normalization:** MRI scans should be normalized to have similar intensity ranges, enhancing model training efficiency.

In [ ]:
```
def load_nii_to_np(folder_path, new_shape=(96, 96, 96)):
    # List to hold all MRI scans
    scans = []

    # Loop through each file in the folder
    for filename in os.listdir(folder_path):
        if filename.endswith('.nii'):
            file_path = os.path.join(folder_path, filename)

            # Load the MRI scan
            scan = nib.load(file_path)

            # Convert the scan into a numpy array
            scan_data = scan.get_fdata().astype(np.float32)
            scan_data = scan_data[:, 50:90, :] #[:, 50:80, :]

            # Normalize the scan data
            normalized_scan = (scan_data - np.min(scan_data)) / (np.max(scan_data)
            data_shape = normalized_scan.shape

            # Resize the scan to add a dimension for grayscale channel
            resized_scan = np.expand_dims(normalized_scan, axis=-1)

            # Add the preprocessed scan to the list
            scans.append(resized_scan)

    # Stack all scans into a single np stack
    scans_np = np.stack(scans)
```

```
    return scans_np, data_shape

# Path to the folder containing the MRI scans
mri_scan_folder_path = "C:\\Users\\chara\\OneDrive\\Desktop\\DM\\dataset\\n171_smwp

# Load the MRI scans into a np stack
mri_scans_np, data_shape = load_nii_to_np(mri_scan_folder_path)
data_shape += (1, )

# Display the shape of the stack
print(mri_scans_np.shape)

print(data_shape)
```

```
(171, 113, 40, 113, 1)
(113, 40, 113, 1)
```

Splitting the data into train-test and then normalizing it

In [ ]:
```
labels_np = threshold_labels[['PT4000', 'PT500']].to_numpy()

# Perform the train-test split
X_train, X_test, y_train, y_test = train_test_split(
    mri_scans_np, labels_np, test_size=0.2, random_state=42
)

# Data augmentation by simply stacking the data 5 times
X_train = np.concatenate([X_train, X_train, X_train, X_train, X_train], axis=0)
y_train = np.concatenate([y_train, y_train, y_train, y_train, y_train], axis=0)

# Flatten the 3D MRI images into 1D vectors
X_train_flattened = X_train.reshape(X_train.shape[0], -1)
X_test_flattened = X_test.reshape(X_test.shape[0], -1)
```

Model Development

MODEL 1: Convolutional Neural Network

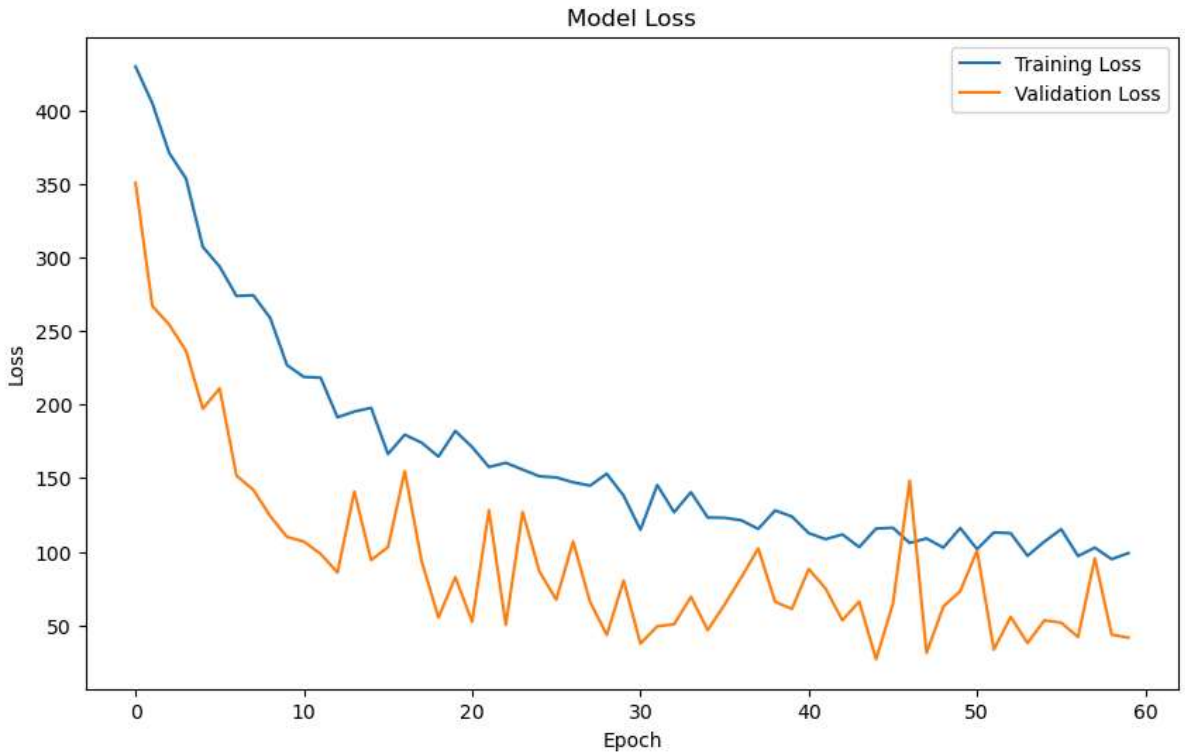
In [ ]:
```
def build_cnn_model(input_shape):
    model = Sequential([
        Input(shape=input_shape),

        Conv3D(64, kernel_size=(3, 3, 3), activation='relu', kernel_initializer='he
        BatchNormalization(),
        MaxPooling3D(pool_size=(2, 2, 2)),

        Conv3D(128, kernel_size=(3, 3, 3), activation='relu', kernel_initializer='h
        BatchNormalization(),
        MaxPooling3D(pool_size=(2, 2, 2)),

        Conv3D(256, kernel_size=(3, 3, 3), activation='relu', kernel_initializer='h
        BatchNormalization(),
        MaxPooling3D(pool_size=(2, 2, 2)),

        Flatten(),
        Dense(256, activation='relu',  kernel_regularizer=l2(0.03)),
        Dropout(0.5),

        Dense(128, activation='relu',  kernel_regularizer=l2(0.03)),
        Dropout(0.5),
```

```
        Dense(2, activation='linear')
    ])

    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=1e-5,
        decay_steps=10000,
        decay_rate=0.9)

    model.compile(optimizer=Adam(learning_rate=lr_schedule), loss='mean_squared_err
    return model

# Convert the datasets and labels into TensorFlow Dataset objects
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
test_dataset = tf.data.Dataset.from_tensor_slices((X_test, y_test))

# Define a batch size
batch_size = 1

# Batch the training dataset
train_dataset = train_dataset.batch(batch_size)
test_dataset = test_dataset.batch(batch_size)

# Calculate the number of batches to use for validation
val_size = int(0.2 * len(X_train) / batch_size)

# Create training and validation datasets
val_dataset = train_dataset.take(val_size)
train_dataset = train_dataset.skip(val_size)

# Create and compile the CNN model
cnn_model = build_cnn_model(data_shape)  # Including the channel dimension

# cnn_model, early_stopping_callback = build_cnn_model(input_shape=(data_shape))

# Fit the model to the training data using the dataset loader
history = cnn_model.fit(train_dataset, validation_data=val_dataset, epochs=60)

# Plotting the training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
plt.show()
```

```
Epoch 1/60
544/544 [==============================] - 52s 76ms/step - loss: 429.7457 - val_lo
ss: 350.7896
Epoch 2/60
544/544 [==============================] - 41s 75ms/step - loss: 404.8020 - val_lo
ss: 266.9575
Epoch 3/60
544/544 [==============================] - 41s 75ms/step - loss: 371.0179 - val_lo
ss: 254.3205
Epoch 4/60
544/544 [==============================] - 42s 77ms/step - loss: 353.6087 - val_lo
ss: 236.5712
Epoch 5/60
544/544 [==============================] - 41s 76ms/step - loss: 307.1413 - val_lo
ss: 197.1069
Epoch 6/60
544/544 [==============================] - 41s 76ms/step - loss: 293.7420 - val_lo
ss: 210.9852
Epoch 7/60
544/544 [==============================] - 41s 76ms/step - loss: 273.7940 - val_lo
ss: 151.6053
Epoch 8/60
544/544 [==============================] - 41s 76ms/step - loss: 274.2768 - val_lo
ss: 142.0759
Epoch 9/60
544/544 [==============================] - 41s 76ms/step - loss: 258.9000 - val_lo
ss: 124.1937
Epoch 10/60
544/544 [==============================] - 41s 76ms/step - loss: 226.7891 - val_lo
ss: 110.0082
Epoch 11/60
544/544 [==============================] - 42s 77ms/step - loss: 218.7768 - val_lo
ss: 106.8774
Epoch 12/60
544/544 [==============================] - 41s 76ms/step - loss: 218.2327 - val_lo
ss: 98.4989
Epoch 13/60
544/544 [==============================] - 42s 76ms/step - loss: 191.3131 - val_lo
ss: 85.7580
Epoch 14/60
544/544 [==============================] - 41s 76ms/step - loss: 195.2258 - val_lo
ss: 140.8307
Epoch 15/60
544/544 [==============================] - 41s 75ms/step - loss: 197.7328 - val_lo
ss: 94.2880
Epoch 16/60
544/544 [==============================] - 41s 76ms/step - loss: 166.3652 - val_lo
ss: 103.0341
Epoch 17/60
544/544 [==============================] - 41s 76ms/step - loss: 179.5117 - val_lo
ss: 154.7297
Epoch 18/60
544/544 [==============================] - 41s 76ms/step - loss: 174.0584 - val_lo
ss: 93.5773
Epoch 19/60
544/544 [==============================] - 41s 76ms/step - loss: 164.5802 - val_lo
ss: 55.2176
Epoch 20/60
544/544 [==============================] - 41s 76ms/step - loss: 182.0165 - val_lo
ss: 82.5799
Epoch 21/60
544/544 [==============================] - 41s 76ms/step - loss: 171.1235 - val_lo
ss: 52.3619
Epoch 22/60
```

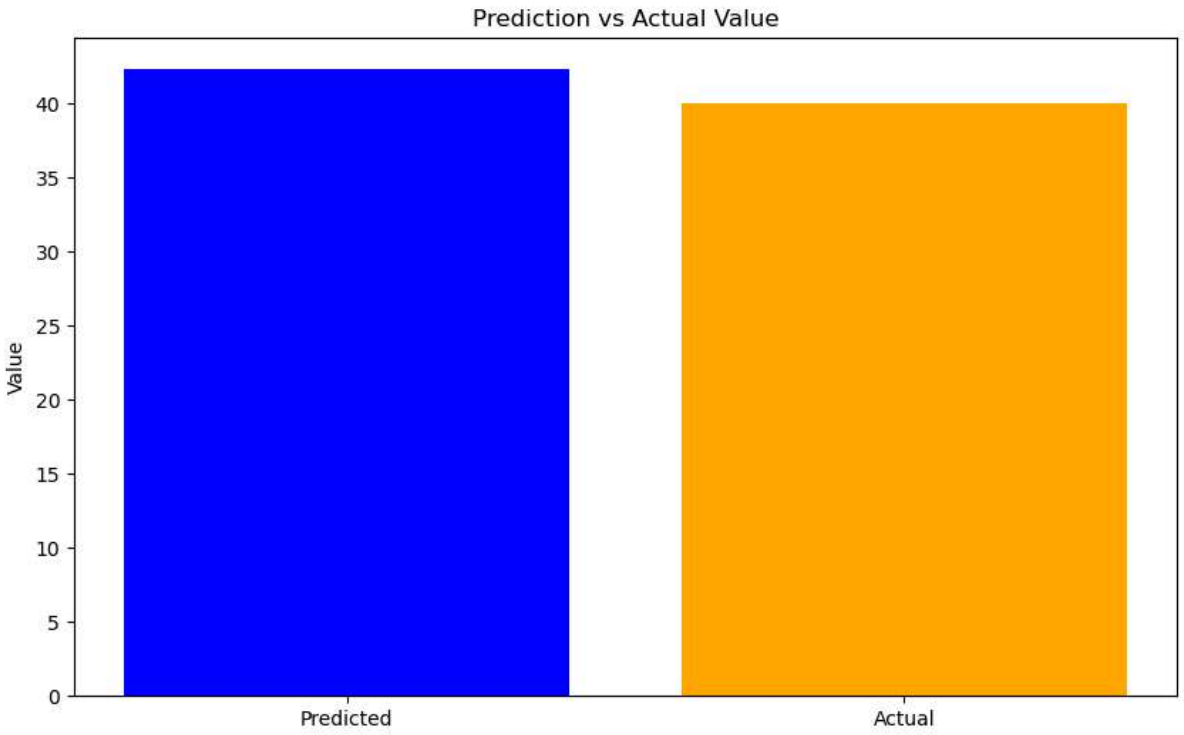
```
544/544 [==============================] - 41s 76ms/step - loss: 157.5156 - val_lo
ss: 128.1856
Epoch 23/60
544/544 [==============================] - 41s 76ms/step - loss: 160.4119 - val_lo
ss: 50.2128
Epoch 24/60
544/544 [==============================] - 41s 76ms/step - loss: 155.7711 - val_lo
ss: 127.0498
Epoch 25/60
544/544 [==============================] - 41s 75ms/step - loss: 151.2218 - val_lo
ss: 86.4102
Epoch 26/60
544/544 [==============================] - 41s 75ms/step - loss: 150.4619 - val_lo
ss: 67.4109
Epoch 27/60
544/544 [==============================] - 41s 76ms/step - loss: 147.1351 - val_lo
ss: 106.8999
Epoch 28/60
544/544 [==============================] - 41s 76ms/step - loss: 144.8862 - val_lo
ss: 66.0865
Epoch 29/60
544/544 [==============================] - 41s 76ms/step - loss: 152.9412 - val_lo
ss: 43.3912
Epoch 30/60
544/544 [==============================] - 41s 75ms/step - loss: 138.0651 - val_lo
ss: 80.3533
Epoch 31/60
544/544 [==============================] - 41s 76ms/step - loss: 114.9008 - val_lo
ss: 37.5195
Epoch 32/60
544/544 [==============================] - 41s 76ms/step - loss: 145.2752 - val_lo
ss: 49.2657
Epoch 33/60
544/544 [==============================] - 42s 76ms/step - loss: 126.7767 - val_lo
ss: 50.6790
Epoch 34/60
544/544 [==============================] - 42s 77ms/step - loss: 140.4059 - val_lo
ss: 69.3204
Epoch 35/60
544/544 [==============================] - 41s 76ms/step - loss: 123.1988 - val_lo
ss: 46.6738
Epoch 36/60
544/544 [==============================] - 41s 75ms/step - loss: 123.0125 - val_lo
ss: 64.0344
Epoch 37/60
544/544 [==============================] - 41s 76ms/step - loss: 121.2800 - val_lo
ss: 82.5888
Epoch 38/60
544/544 [==============================] - 41s 76ms/step - loss: 115.4524 - val_lo
ss: 102.1727
Epoch 39/60
544/544 [==============================] - 41s 76ms/step - loss: 127.9981 - val_lo
ss: 65.9504
Epoch 40/60
544/544 [==============================] - 41s 75ms/step - loss: 123.8809 - val_lo
ss: 61.0738
Epoch 41/60
544/544 [==============================] - 41s 75ms/step - loss: 112.5949 - val_lo
ss: 88.2301
Epoch 42/60
544/544 [==============================] - 41s 76ms/step - loss: 108.4489 - val_lo
ss: 74.9400
Epoch 43/60
544/544 [==============================] - 41s 76ms/step - loss: 111.7021 - val_lo
```

```
ss: 53.3528
Epoch 44/60
544/544 [==============================] - 41s 76ms/step - loss: 103.0751 - val_lo
ss: 66.0195
Epoch 45/60
544/544 [==============================] - 41s 76ms/step - loss: 115.6727 - val_lo
ss: 26.8988
Epoch 46/60
544/544 [==============================] - 41s 76ms/step - loss: 116.2183 - val_lo
ss: 64.4669
Epoch 47/60
544/544 [==============================] - 41s 76ms/step - loss: 105.9053 - val_lo
ss: 148.2113
Epoch 48/60
544/544 [==============================] - 41s 76ms/step - loss: 108.9850 - val_lo
ss: 31.1357
Epoch 49/60
544/544 [==============================] - 41s 76ms/step - loss: 102.7333 - val_lo
ss: 62.7707
Epoch 50/60
544/544 [==============================] - 41s 76ms/step - loss: 116.0024 - val_lo
ss: 73.0691
Epoch 51/60
544/544 [==============================] - 41s 76ms/step - loss: 101.6987 - val_lo
ss: 100.6970
Epoch 52/60
544/544 [==============================] - 41s 75ms/step - loss: 113.0789 - val_lo
ss: 33.4150
Epoch 53/60
544/544 [==============================] - 40s 74ms/step - loss: 112.5414 - val_lo
ss: 55.6788
Epoch 54/60
544/544 [==============================] - 41s 74ms/step - loss: 97.1905 - val_los
s: 37.8785
Epoch 55/60
544/544 [==============================] - 40s 74ms/step - loss: 106.8001 - val_lo
ss: 53.2741
Epoch 56/60
544/544 [==============================] - 41s 75ms/step - loss: 115.2657 - val_lo
ss: 51.7376
Epoch 57/60
544/544 [==============================] - 41s 75ms/step - loss: 97.0272 - val_los
s: 41.9537
Epoch 58/60
544/544 [==============================] - 41s 75ms/step - loss: 102.6809 - val_lo
ss: 95.4467
Epoch 59/60
544/544 [==============================] - 41s 75ms/step - loss: 94.9198 - val_los
s: 43.5791
Epoch 60/60
544/544 [==============================] - 41s 75ms/step - loss: 99.0350 - val_los
s: 41.4865
```



In [ ]:
```
# Assuming ny_index is the index of the NY value in the X_test set
ny_index = 2

# Replace 0 with the actual index of the NY value you're interested in

# Select the specific test sample and its actual label
ny_sample = X_test[ny_index]
ny_actual_label = y_test[ny_index]

# Since the model expects a batch, we need to add an extra dimension
ny_sample_expanded = np.expand_dims(ny_sample, axis=0)

# Predict using the model
ny_prediction = cnn_model.predict(ny_sample_expanded)

# Print the predicted and actual values
print(f"Predicted value: {ny_prediction.flatten()}")
print(f"Actual value: {ny_actual_label}")

# Visualizing the prediction and actual value
plt.figure(figsize=(10, 6))
plt.bar(['Predicted', 'Actual'], [ny_prediction.flatten()[0], ny_actual_label[0]],
plt.ylabel('Value')
plt.title('Prediction vs Actual Value')
plt.show()
```

```
1/1 [==============================] - 1s 604ms/step
Predicted value: [42.30235  11.128555]
Actual value: [40 18]
```



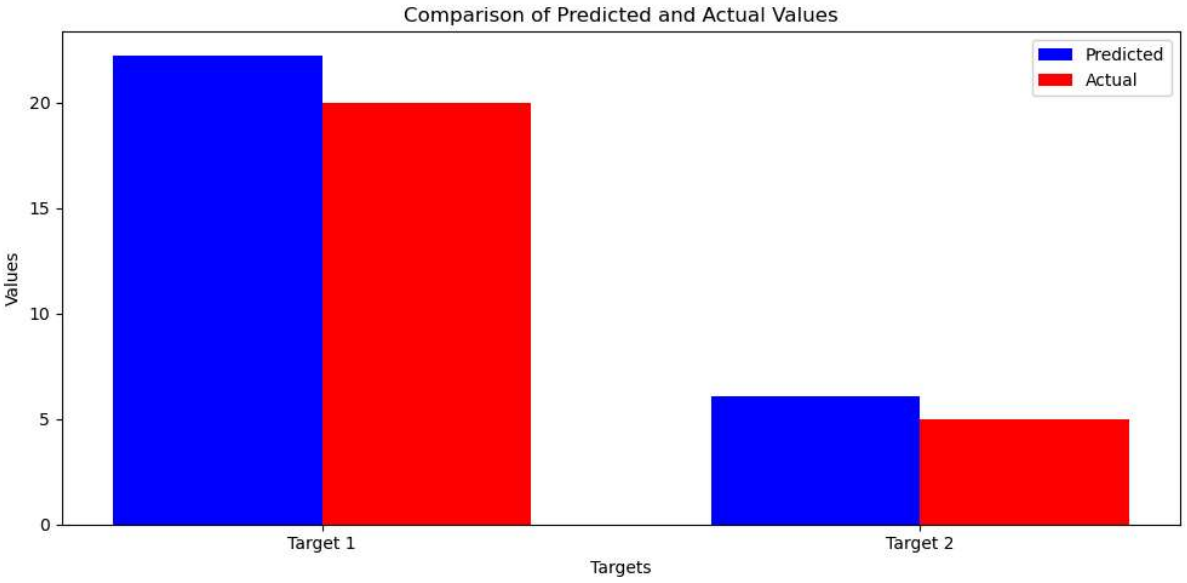
MODEL 2: Support Vector Machine (SVM) with PCA Script

In [ ]:
```
def train_svm_with_pca_per_target(X, y, n_components=10):
    # y should be of shape [n_samples, n_targets]
    models = []
    for i in range(y.shape[1]):  # Assuming y has shape [n_samples, n_targets]
        pipeline = make_pipeline(PCA(n_components=n_components), SVR(C=1.0, epsilor
        pipeline.fit(X, y[:, i])
        models.append(pipeline)
    return models

# Train a separate model for each target
svm_model = train_svm_with_pca_per_target(X_train_flattened, y_train)
```

In [ ]:
```
# Assuming ny_index is the index of the specific test value you're interested in
ny_index = 3  # You can replace 3 with any valid index from your test set

# Select the specific test sample
ny_sample = X_test_flattened[ny_index]  # Ensure this is the flattened version if t

# Predict using each model. Since each model expects a single instance to be a 2D a
ny_predictions = [model.predict(ny_sample.reshape(1, -1))[0] for model in svm_model

# Assuming you have the actual labels for comparison
ny_actual_labels = y_test[ny_index]

# Print the predicted and actual values
print("Predicted values:", ny_predictions)
print("Actual values:", ny_actual_labels)

# Number of targets
num_targets = len(ny_predictions)

# Plotting predicted vs actual values
plt.figure(figsize=(10, 5))
bar_width = 0.35
index = np.arange(num_targets)

plt.bar(index, ny_predictions, bar_width, label='Predicted', color='b')
```

```
plt.bar(index + bar_width, ny_actual_labels, bar_width, label='Actual', color='r')

plt.xlabel('Targets')
plt.ylabel('Values')
plt.title('Comparison of Predicted and Actual Values')
plt.xticks(index + bar_width / 2, ('Target 1', 'Target 2'))  # Adjust labels as nee
plt.legend()

plt.tight_layout()
plt.show()
```

```
Predicted values: [22.243050830990008, 6.094474995437929]
Actual values: [20  5]
```



MODEL 3: XGBOOST

In [ ]:

```
# Feature selection for dimensionality reduction
def select_features(X_train, y_train, X_test, k=10):

    # Create empty masks to combine feature selection from each target
    combined_mask = np.zeros(X_train.shape[1], dtype=bool)

    # Perform feature selection for each target variable
    for i in range(y_train.shape[1]):
        # Select the best k features for the i-th target
        selector = SelectKBest(score_func=f_regression, k=k)
        X_new = selector.fit_transform(X_train, y_train[:, i])

        # Combine the masks (logical OR)
        combined_mask = combined_mask | selector.get_support()

    # Apply the combined mask to the training and testing sets
    X_train_selected = X_train[:, combined_mask]
    X_test_selected = X_test[:, combined_mask]

    return X_train_selected, X_test_selected, combined_mask

# Perform feature selection
X_train_selected, X_test_selected, feature_mask = select_features(
    X_train_flattened, y_train, X_test_flattened, k=9
)

# Define a set of hyperparameters to tune
param_grid = {
    'estimator__max_depth': [3, 4, 5],
```

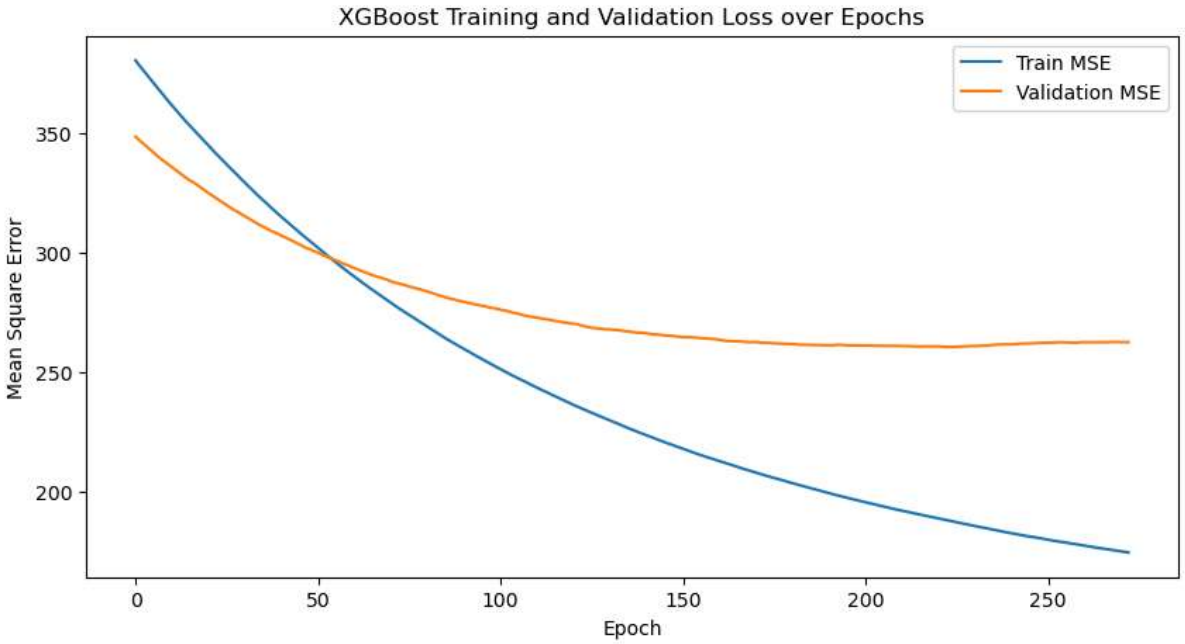
```
    'estimator__n_estimators': [50, 100, 200],
    'estimator__learning_rate': [0.01, 0.1, 0.2],
    'estimator__subsample': [0.7, 0.8, 0.9],
    'estimator__colsample_bytree': [0.7, 0.8, 0.9],
}

# Set up the XGBoost model inside MultiOutputRegressor
xgboost_model = MultiOutputRegressor(XGBRegressor(objective='reg:squarederror'))

# Set up GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(estimator=xgboost_model, param_grid=param_grid, cv=3, n_

# Fit GridSearchCV to the training data
grid_search.fit(X_train_selected, y_train)  # Use X_train_selected if you did featu

# Best hyperparameters
print("Best hyperparameters:", grid_search.best_params_)

# Best model
xgboost_model = grid_search.best_estimator_
```

```
Fitting 3 folds for each of 243 candidates, totalling 729 fits
Best hyperparameters: {'estimator__colsample_bytree': 0.8, 'estimator__learning_ra
te': 0.2, 'estimator__max_depth': 5, 'estimator__n_estimators': 200, 'estimator__s
ubsample': 0.7}
```

In [ ]:

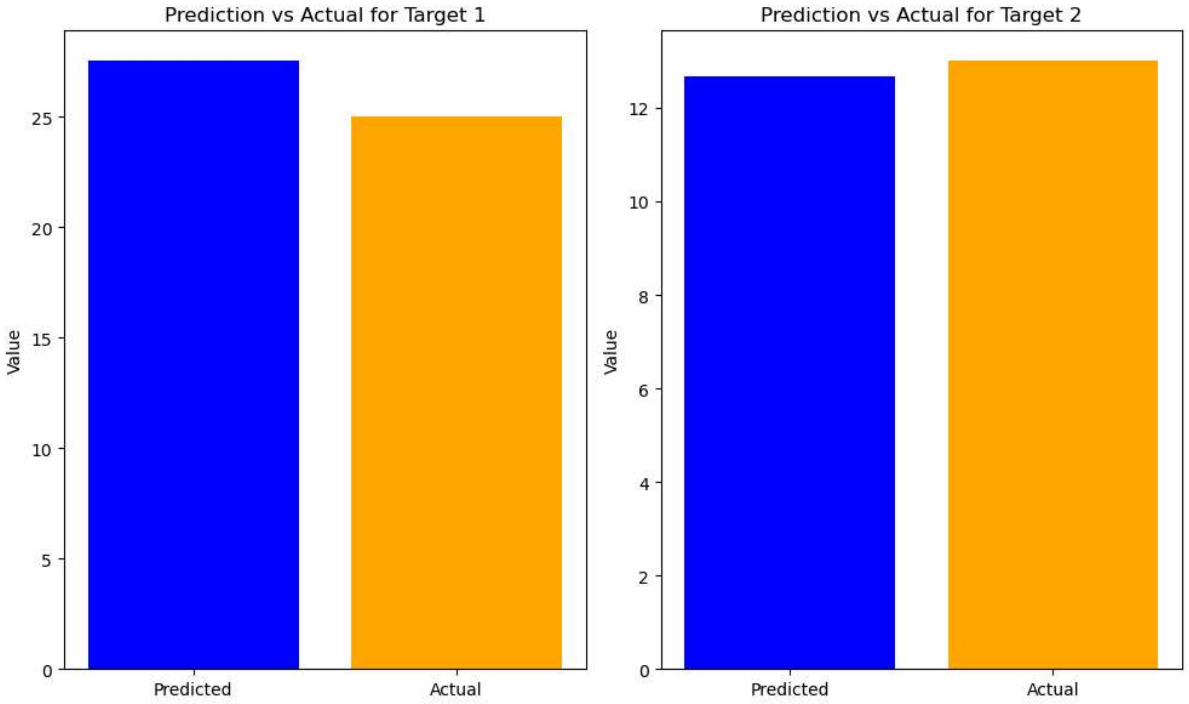
```
from xgboost import XGBRegressor, DMatrix, train
import matplotlib.pyplot as plt
import math

# Assuming you already have X_train_selected, y_train, X_test_selected, y_test prop
# Create DMatrix for train and validation
dtrain = DMatrix(X_train_selected, label=y_train)
dval = DMatrix(X_test_selected, label=y_test)  # Here using test data as validatior

# Define parameters (using previously identified or default reasonable values)
params = {
    'max_depth': 2,  # Reduced depth to simplify each tree
    'learning_rate': 0.005,  # Further reduced to ensure more gradual learning
    'subsample': 0.6,  # Reduced to lower the fraction of samples used per tree
    'colsample_bytree': 0.6,  # Reduced to lower the fraction of features used per
    'objective': 'reg:squarederror',
    'reg_alpha': 0.05,  # Increased L1 regularization to promote sparsity
    'reg_lambda': 2.5,  # Increased L2 regularization to add more penalty on model
    'booster': 'dart'  # DART booster to help reduce overfit
}

# Add early stopping
evals_result = {}
model = train(params, dtrain, num_boost_round=2000, evals=[(dtrain, 'train'), (dval
              early_stopping_rounds=50, evals_result=evals_result, verbose_eval=Fal

# Extract the training and validation loss
epochs = len(evals_result['train']['rmse'])
x_axis = range(0, epochs)
train_rmse = evals_result['train']['rmse']
val_rmse = evals_result['eval']['rmse']

train_mse = [value ** 2 for value in train_rmse]
val_mse = [value ** 2 for value in val_rmse]

# Plotting the training and validation loss graph
plt.figure(figsize=(10, 5))
plt.plot(x_axis, train_mse, label='Train MSE')
```

```
plt.plot(x_axis, val_mse, label='Validation MSE')
plt.title('XGBoost Training and Validation Loss over Epochs')
plt.legend()
plt.ylabel('Mean Square Error')
plt.xlabel('Epoch')
plt.show()
```



In [ ]:
```
# Assuming `ny_index` is the index of the specific test value you're interested in
ny_index = 8  # You can replace 3 with any valid index from your test set

# Select the specific test sample and its actual labels
ny_sample = X_test_selected[ny_index]
ny_actual_label = y_test[ny_index]

# Since the model doesn't require the input to have a batch dimension,
# you should still ensure the input is two-dimensional (1, number of features)
ny_sample_expanded = ny_sample.reshape(1, -1)

# Predict using the model
ny_prediction = xgboost_model.predict(ny_sample_expanded)

# Print the predicted and actual values
print("Predicted value:", ny_prediction[0])
print("Actual value:", ny_actual_label)

# Visualizing the prediction and actual value for each target
plt.figure(figsize=(10, 6))
num_targets = ny_prediction.shape[1]  # Should be 2 based on your setup
for i in range(num_targets):
    plt.subplot(1, num_targets, i + 1)
    plt.bar(['Predicted', 'Actual'], [ny_prediction[0][i], ny_actual_label[i]], col
    plt.title(f'Prediction vs Actual for Target {i+1}')
    plt.ylabel('Value')
plt.tight_layout()
plt.show()
```

```
Predicted value: [27.52304  12.680429]
Actual value: [25 13]
```



Evaluation

In [ ]:

```python
def evaluate_model(model, X_test, y_test, model_type='cnn'):
    """
    Evaluate the given model on the test set.

    Parameters:
    - model: The trained model (CNN, SVM pipeline list, or XGBoost).
    - X_test: Test set images or features.
    - y_test: True hearing thresholds for the test set.
    - model_type: Type of the model ('cnn', 'svm', or 'xgboost').

    Returns:
    - MAE: Mean Absolute Error of the predictions.
    - MSE: Root Mean Squared Error of the predictions.
    """
    # Initialize predictions array
    predictions = None

    # Preprocess test data based on model type
    if model_type == 'cnn':
        predictions = model.predict(X_test)

    elif model_type == 'svm':
        # Ensure X_test is appropriately shaped (flattened if needed)
        if len(X_test.shape) > 2:
            X_test_flattened = np.reshape(X_test, (X_test.shape[0], -1))
        else:
            X_test_flattened = X_test

        # Aggregate predictions from each model in the list
        predictions = np.column_stack([mod.predict(X_test_flattened) for mod in mod

    elif model_type == 'xgboost':
        predictions = model.predict(X_test)

    else:
        raise ValueError("Unsupported model type. Choose 'cnn', 'svm', or 'xgboost'

    # Compute evaluation metrics
```

```
    mae = mean_absolute_error(y_test, predictions)
    mse = mean_squared_error(y_test, predictions)  # Use squared=False if you need

    return mae, mse

test_dataset = tf.data.Dataset.from_tensor_slices((X_test))
test_dataset = test_dataset.batch(batch_size)

mae, mse = evaluate_model(cnn_model, test_dataset, y_test, model_type='cnn')
print(f"CNN Model - MAE: {mae}, MSE: {mse}")

mae, mse = evaluate_model(svm_model, X_test_flattened, y_test, model_type='svm')
print(f"SVM Model - MAE: {mae}, MSE: {mse}")

mae, mse = evaluate_model(xgboost_model, X_test_selected, y_test, model_type='xgboo
print(f"xgboost Model - MAE: {mae}, MSE: {mse}")
```

```
35/35 [==============================] - 1s 20ms/step
CNN Model - MAE: 11.23733310018267, MSE: 230.08263556371926
SVM Model - MAE: 12.459703497190832, MSE: 244.3327747137499
xgboost Model - MAE: 14.381235516071321, MSE: 338.47099563145787
```

Code to visualize the data:

```
import nibabel as nib
import matplotlib.pyplot as plt

# Load the .nii file
nii_path = 'path_to_your_nii_file.nii'
img = nib.load(nii_path)

# Get the data as a numpy array
data = img.get_fdata()

# Function to display a slice
def show_slice(slice):
   plt.imshow(slice.T, cmap="gray", origin="lower")
   plt.axis('off')

# Displaying slices from the middle of each dimension
fig, axes = plt.subplots(1, 3)
slice_0 = data[data.shape[0] // 2, :, :]
slice_1 = data[:, data.shape[1] // 2, :]
slice_2 = data[:, :, data.shape[2] // 2]
show_slice(slice_0)
axes[0].set_title('Slice X')
show_slice(slice_1)
axes[1].set_title('Slice Y')
show_slice(slice_2)
axes[2].set_title('Slice Z')

# Show the plots
plt.show()
```