Artificial Image Generation for ASL Image Recognition
Summer 2024

Liam O'Donnell, Hoang Nguyen, Gedeng Ruan

1 Roles and Responsibilities

Liam O'Donnell

- Dataset selection
- Assembly of train and test datasets
- Artificial image generation for class balancing
- Initial model construction

Hoang Nguyen

- Dataset selection and uploading
- Repository manager
- Training and Tuning of Neural Network

Gedeng Ruan

- Hyper parameter tuning and construction of the following classification models:
  - XGBoost
  - KNN
  - Naive Bayes
  - Random Forest

2 Problem Statement

According to the Commission on the Deaf and Hard of Hearing, approximately 500,000 people in the United States use American Sign Language (ASL) as their native language [1]. However, a significant communication barrier exists because the other 329.5 million Americans generally lack the knowledge and understanding of ASL to communicate effectively. This project aims to address this issue by developing a machine learning model that can accurately interpret and classify ASL signs from color images.

The primary goal of the group's research is to establish a foundation for ASL sign language interpretation and classification. By testing multiple machine learning models, the group aimed to determine the most accurate and efficient model for classifying the alphanumeric characters represented in ASL images.

During the ISYE 6740 course, a key focus has been on image classification using the MNIST hand-written digits dataset. This experience has provided valuable insights into how computers process images. However, the MNIST dataset consists of two classes of pixel values and low-resolution images, which do not capture the complexity required for effective ASL image classification. The group's project necessitates more advanced model training and tuning to account for the variability in hand shapes, photo contrast, and skin tones.

To improve model performance and balance the classes within our dataset, the group generated artificial images. This approach allowed the group to address the inherent challenges and enhance the robustness of our classification algorithm.

3 Data Source

3.1 Original Dataset

The group's original models were built off of data retrieved from https://www.kaggle.com/datasets/rahulmakwana/american-sign-language-recognition [2]. As shown in **Figure 1**, several issues arose from using this data set.

1. Images appear binary in color (B/W)

2. Testing and Training data shape and size in nature, often only differing by a couple of pixels.

(a) Original Data Source - Number 1 (Training)

(b) Original Data Source - Number 1 (Testing)

Figure 1: Comparison of Training and Testing Data Sources for Number 1

While the data set size was ideal for training and fine tuning a model (50,000+ training images, 7,000 testing images), the group recognized the fact that this model would not perform optimally when image data that contained alternative colors was introduced (i.e. greyscale). The group accepted the hypothesis that any models built using this data would produce an overfit result by constructing KNN, KMeans, and RandomForest models. All of the aforementioned models achieved 99% accuracy across the testing data set, with no hyperparameter tuning, indicating that the lack in variability in color ranges, and images, had an overall negative impact on the group's goal to create a robust, flexible image classifying tool.

```
Evaluating RandomForestClassifier
Accuracy: 0.9995857841106784
Classification Report:
              precision    recall  f1-score   support

           1       1.00      1.00      1.00       276
          10       1.00      1.00      1.00       310
           2       1.00      1.00      1.00       293
           3       1.00      1.00      1.00       287
           4       1.00      1.00      1.00       300
           5       1.00      1.00      1.00       306
           6       1.00      1.00      1.00       310
           7       1.00      1.00      1.00       319
           8       1.00      1.00      1.00       292
           9       1.00      1.00      1.00       293
           A       1.00      1.00      1.00       302
           B       1.00      1.00      1.00       318
           C       1.00      1.00      1.00       303
           D       1.00      1.00      1.00       319
           E       1.00      0.99      1.00       262
           F       1.00      1.00      1.00       296
           G       1.00      1.00      1.00       305
```

Figure 2: Original Dataset - Model Results

3.2 Revised/Final Dataset

The group used the lessons learned during the model building process in the original dataset, to obtain a final set of images that achieved the following:

- Variability in images - differing positions of the hand sign
- Variability in hand shape/size/skin tone
- Alphanumeric characters
- Color Images

The group conducted the search of a new image set via Kaggle and elected to use "*Hand Sign Images Dataset*" [3] for alphabetic characters and "*American Sign Language Dataset*" [4] for numeric characters. As shown in **Figure 3** and **Figure 4**, these datasets offered a varying range of images, which would ensure our model captured the variability discussed above.

Figure 3: Letter A - Revised/Final Dataset (Alphabetic)

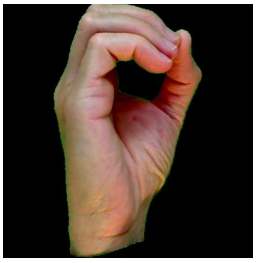


Figure 4: Number 0 - Revised/Final Dataset (Numeric)

Selection of these datasets resulted in several prepossessing steps which will be discussed in depth in the next section.

Note: All further sections in this project paper are built off of the Revised/Final Dataset. There will be no further comment on the original dataset.

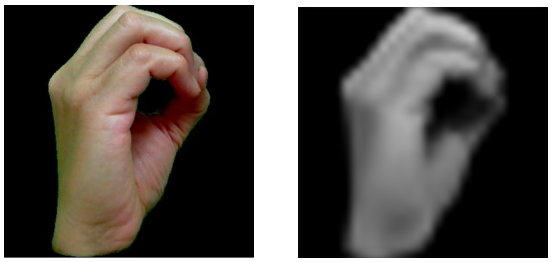
4 Data Prepossessing

Multiple prepossessing steps needed to be taken to ensure proper model training. These steps included:

- Adjusting color of numeric data set to address data leakage.
- Re-scaling numeric dataset to match alphabetic dataset
- Addressing class imbalance between numeric training images and alphabetic images

4.1 Converting Image color and size

A discrepancy between the numeric data set and the alphabetic dataset was image color (numeric was colored images, alphabetic was grey scale) and image size (alphabetic was 28x28, numeric was 400x400). The group elected to convert all images to grey scale and 28x28 to speed up the training process. The transformation can be shown in the below figures.



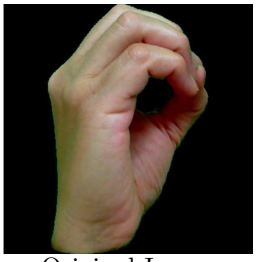
(a) Original Img **(b)** Transformed Img

Figure 5: Image Transformation (Number 0)

4.2 Addressing Class Imbalance and Generation of Artificial Images

While the group's selection of two datasets increased the variety of images available for training models, the issue of class imbalance quickly became apparent. The alphabetic dataset contained approximately 275 training images per character, whereas the numeric dataset had roughly 20 training images per character. This significant discrepancy would likely cause the model to overfit to alphabetic characters, thereby reducing its overall accuracy and robustness.

To address this challenge, the group implemented a data augmentation strategy to artificially increase the number of training images for the numeric dataset. This approach involved generating new images through a series of transformations, including random rotations, brightness adjustments, and image flipping. Each transformation was carefully calibrated to ensure that the newly created images retained their original characteristics while introducing sufficient variability to enhance the training process. For example, random rotations were applied within a specified range of degrees to simulate different orientations of the characters. Brightness adjustments were made to account for variations in lighting conditions, ensuring that the model could generalize well to different environments. Image flipping, both horizontally and vertically, introduced additional diversity to the dataset, helping the model learn to recognize characters from different perspectives. **Figure 6** shows the process of artificial image generation.



Original Image

Image Converted to Grey Scale

Image Resized to (28,28)

Image Randomly Rotated Between [-180, 180] degrees

Final Image - Brightness Randomly Increased By a Random Factor Between [1, 1.5]

Figure 6: Flow Chart of Artificial Image Generation

Through these augmentation techniques, the numeric training dataset was balanced with the alphabetic training dataset, resulting in a more equitable distribution of training images across all classes. This process ultimately produced approximately 2000 new artificial training images, significantly bolstering the dataset's size and diversity.

5 Methodology

Hyperparameter tuning is a important step in optimizing the performance of machine learning models [5]. Properly tuned hyperparameters can significantly improve the model's accuracy, precision, recall, and other performance metrics. Tuning helps find the right bias-variance tradeoff balance, thus preventing overfitting. Efficient hyperparameter tuning can also reduce the training time by avoiding unnecessary computations with suboptimal parameters.

There are multiple approaches to perform hyperparameter tuning. In this report, the group employed grid search and cross-validation random search. While more advanced techniques like Bayesian optimization, which utilize probabilistic models, exist, the chosen methods were deemed suitable for the scope and objectives of this project.

Since each type of ML models is different mathematically, it requires different sets of hyperparameters to be tuned for different models. Here, the group discussed hyperparameter tuning for five different machine learning models: XGBoost, Random Forest, KNN, Naive Bayes, and Neural Networks.

5.1 XGBoost

For XGBoost, the group performed 100 runs of 3-fold CV random search to tune the hyperparameters and used accuracy as the objective function. The hyperparameter grid and results were shown in **Table 1**.

Table 1: Summary of XGBoost hyperparameter Tuning

hyperparameter	**Definition**	**Parameter Grid**	**Optimal Value**
n_estimators	number of boosting rounds	50, 100, 150, 200	200
learning_rate	step size during the boosting process	0.01, 0.029, 0.048, 0.067, 0.086, 0.105, 0.124, 0.143, 0.162, 0.181, 0.2	0.2
max_depth	depth of each tree	3, 4, 5, 6, 7	6
subsample	fraction of samples used for fitting individual trees	0.7, 0.8, 0.9, 1.0	0.7
colsample_bytree	fraction of features used for fitting individual trees	0.7, 0.8, 0.9, 1.0	0.9

Below is the discussion of the impact of each hyperparameter on the XGBoost performance.

Generally, more rounds of boosting can lead to better performance but also increase the risk of overfitting. As shown in **Figure** 7, when the $n\_estimator = 200$, the mean test score is on the high end with the lowest variance. From the random search, 200 is the optimal value for n_estimator. Learning Rate controls the step size during the boosting process. Lower values make the model more robust to overfitting but require more boosting rounds. As shown in **Figure** 8, the general trend is that the mean test score increase when the learning rate increases. Intuitively, relative higher learning rate leads to faster model training if it converges. From the random search, 0.2 is the optimal value for learning rate. Maximum Depth (max_depth) controls the depth of each tree. Deeper trees can capture more complex patterns but are more prone to overfitting. As shown in **Figure** 9, the mean test score reaches plateau when the max_depth increase to 5 or higher. From the random search, 6 is the optimal value for maximum depth. $subsample$ is the fraction of samples

used for fitting individual trees. Lower values prevent overfitting but too low values might underfit. As shown in **Figure** 10, overall 0.8 is the worst for the subsample. To prevent overfitting, 0.7 is selected as the optimal value for *subsample* from the random search. Column Subsample (*colsample_bytree*) is the fraction of features used for fitting individual trees and it helps in reducing overfitting. As shown in **Figure** 11, when $colsample\_bytree = 0.8$, the mean test score is high with lowest variance. From the random search, 0.9 is selected as the optimal value for *colsample_bytree*.

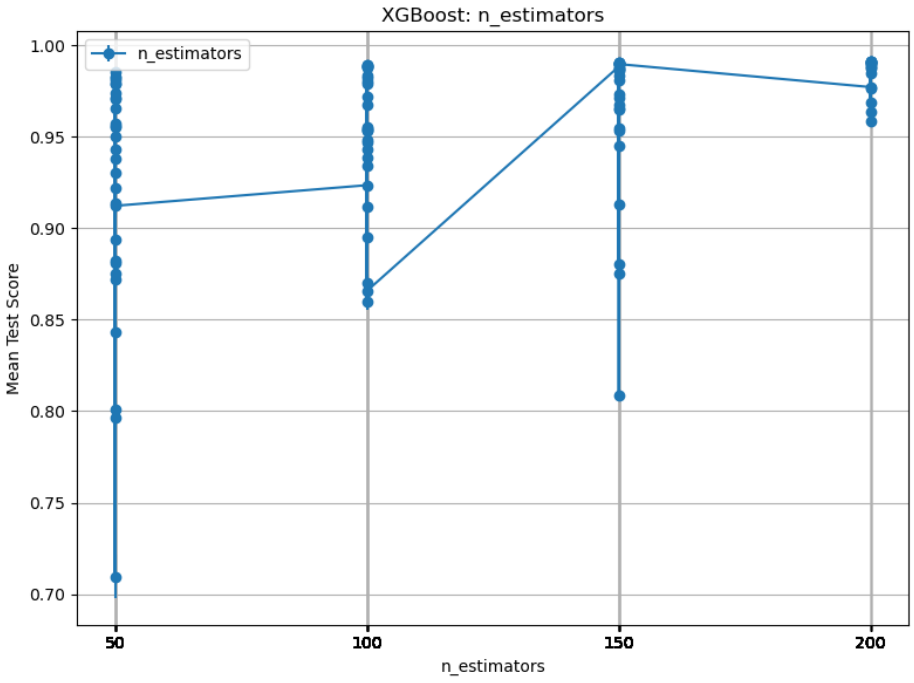


Figure 7: XGBoost Hyperparameter Tuning Results: n_estimators

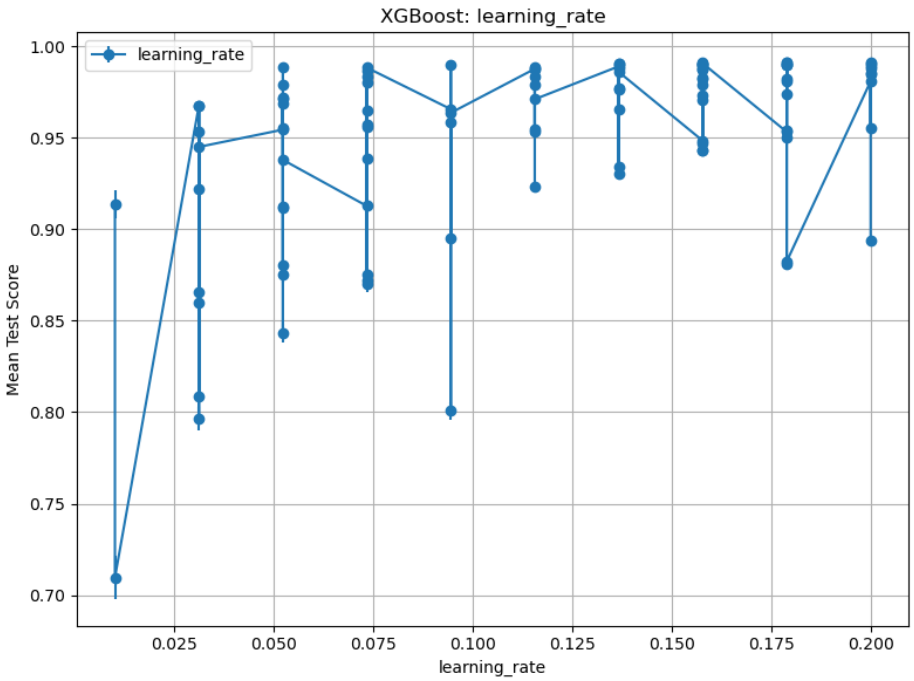


Figure 8: XGBoost Hyperparameter Tuning Results: learning rate

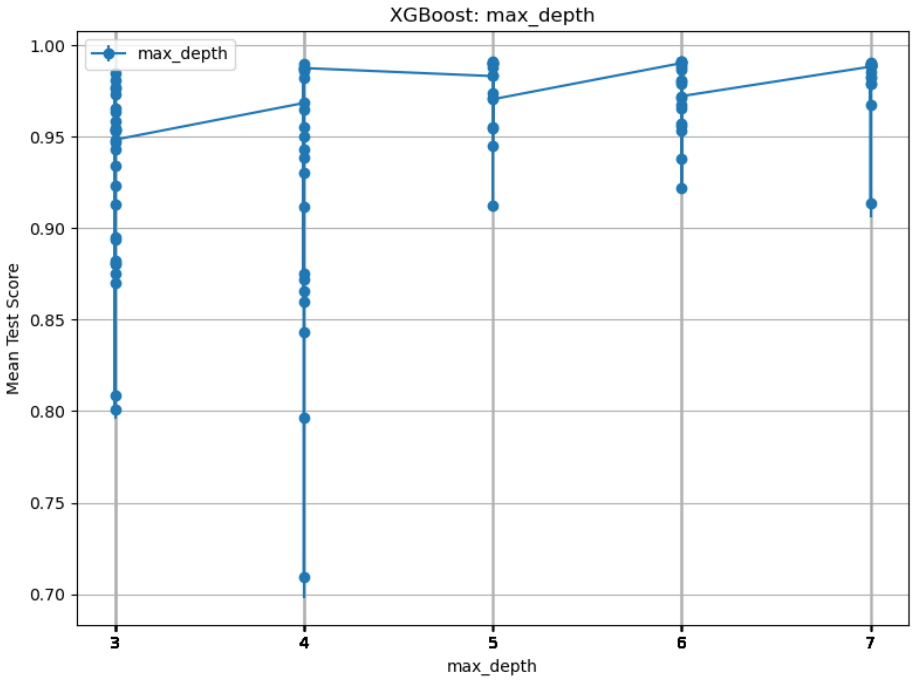


Figure 9: XGBoost Hyperparameter Tuning Results: max depth

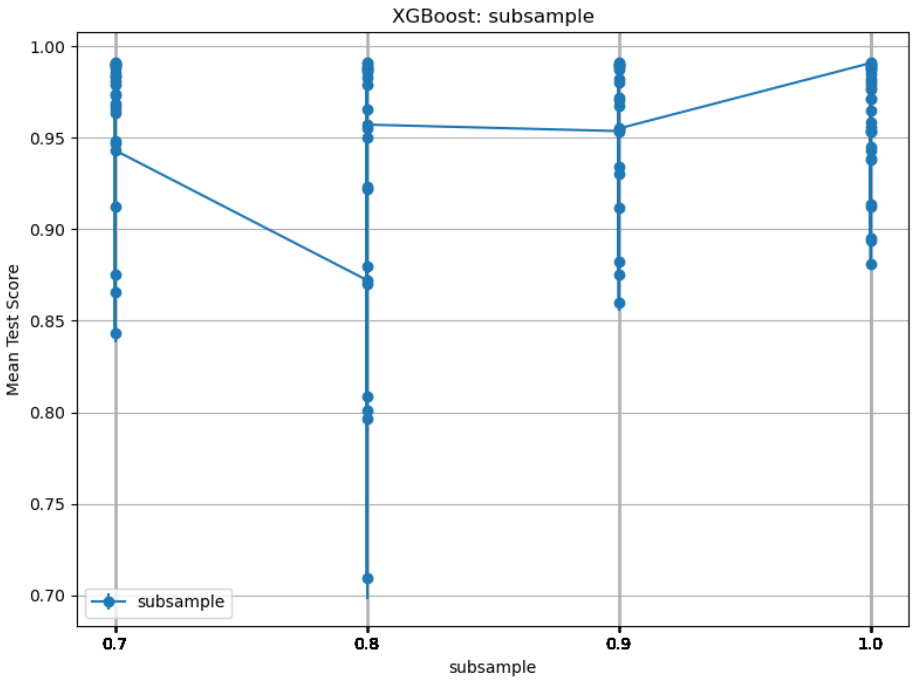


Figure 10: XGBoost Hyperparameter Tuning Results: subsample

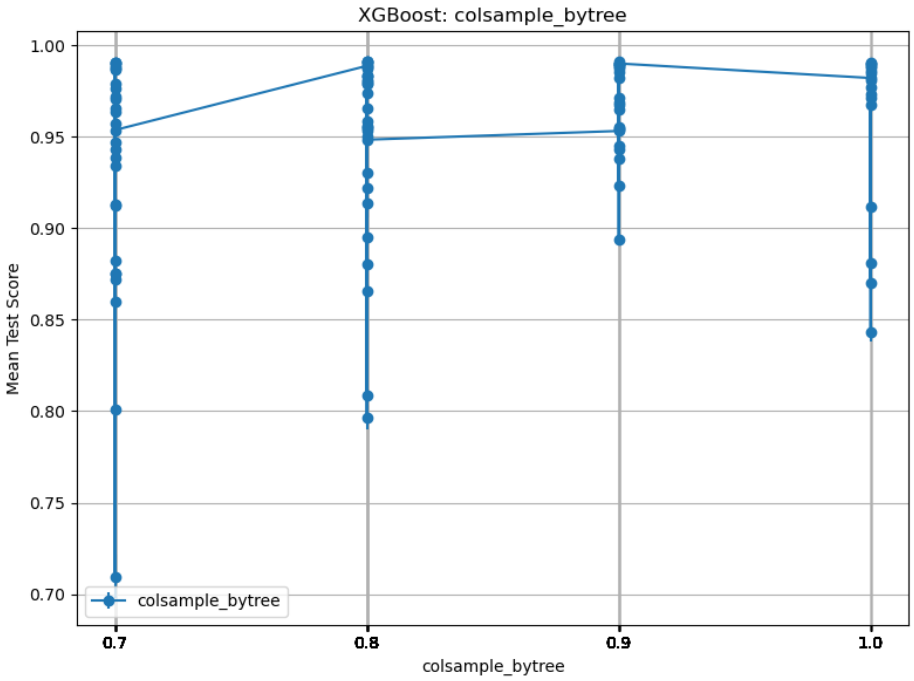


Figure 11: XGBoost Hyperparameter Tuning Results: colsample_bytree

5.2 K-Nearest Neighbors (KNN)

For K-Nearest Neighbors, the group used 3-fold grid search to tune the hyperparameters and used accuracy as the objective function. The hyperparameter grid and results were shown in **Table 2**.

Table 2: Summary of KNN hyperparameter Tuning

hyperparameter	Definition	Parameter Grid	Optimal Value
n_neighbors	number of neighbors	3, 5, 7, 10	3
weights	weight function	uniform, distance	distance
metric	distance metric	euclidean, manhattan	euclidean

The optimal number of neighbors (k) is crucial. Too low can lead to overfitting, while too high can cause underfitting. As shown in **Figure** 12, overall the mean test scores are similar for the tested values. When the number of neighbors $k = 3$, it has the highest mean test score. From the grid search, the optimal number of neighbors $k = 3$. The weight function can help in cases where closer neighbors should have more influence. Since the dataset is 2-D image, this weight hyperparameter does not have big impact, as shown in **Figure** 13. From the grid search, the optimal weight function is distanced base function. The choice of distance metric can significantly impact performance, especially for high-dimensional data. In this project, the group was trying to build classification model for 2-D image dataset, which is not high-dimensional data. Overall the mean test scores are similar for euclidean and manhattan metric, as shown in **Figure** 14. From the grid search, euclidean is the optimal metric for KNN.

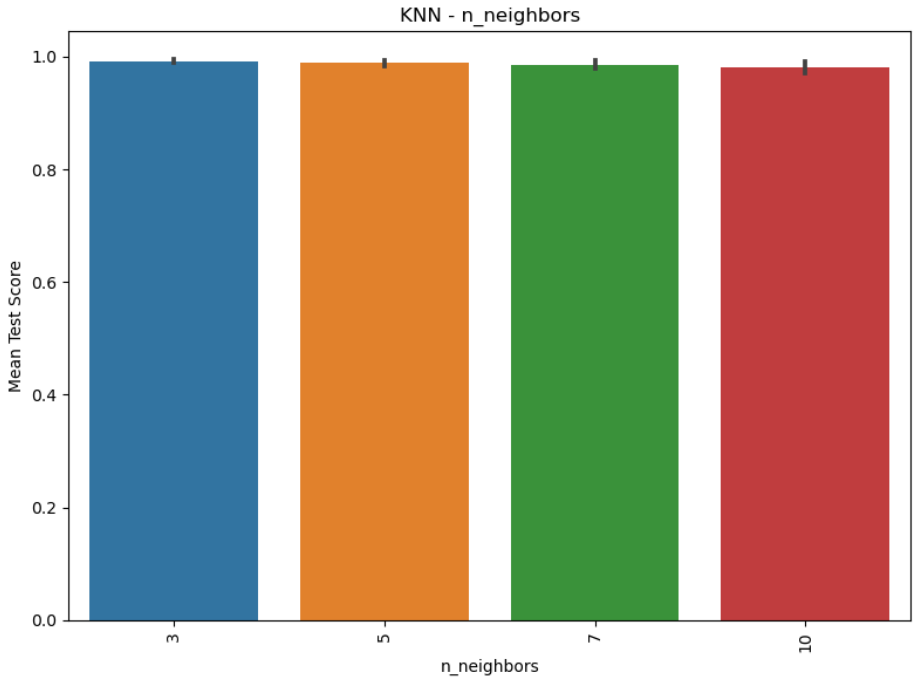


Figure 12: KNN Hyperparameter Tuning Results: n_neighbors

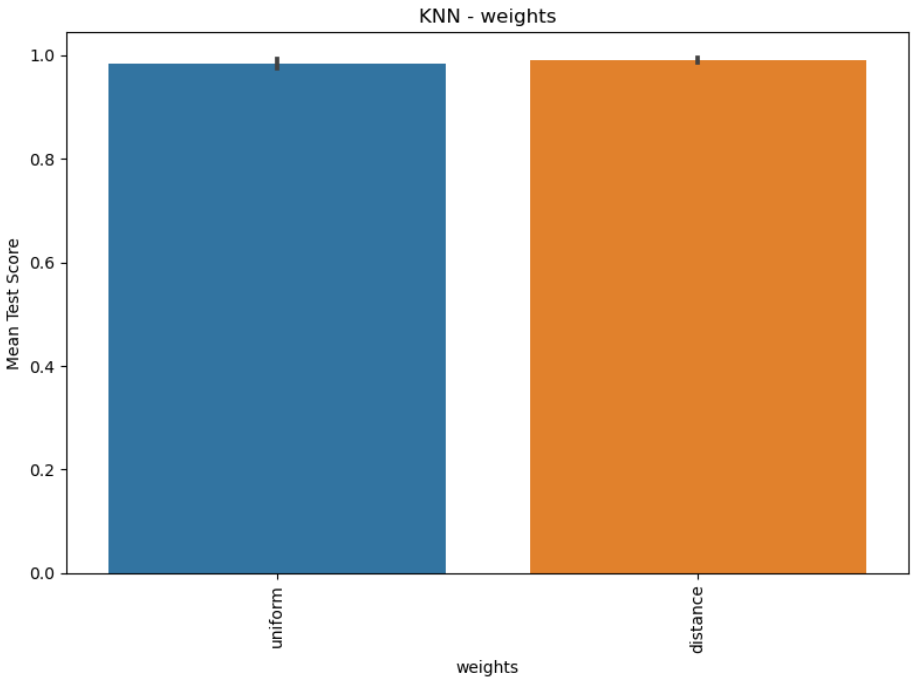


Figure 13: KNN Hyperparameter Tuning Results: weight

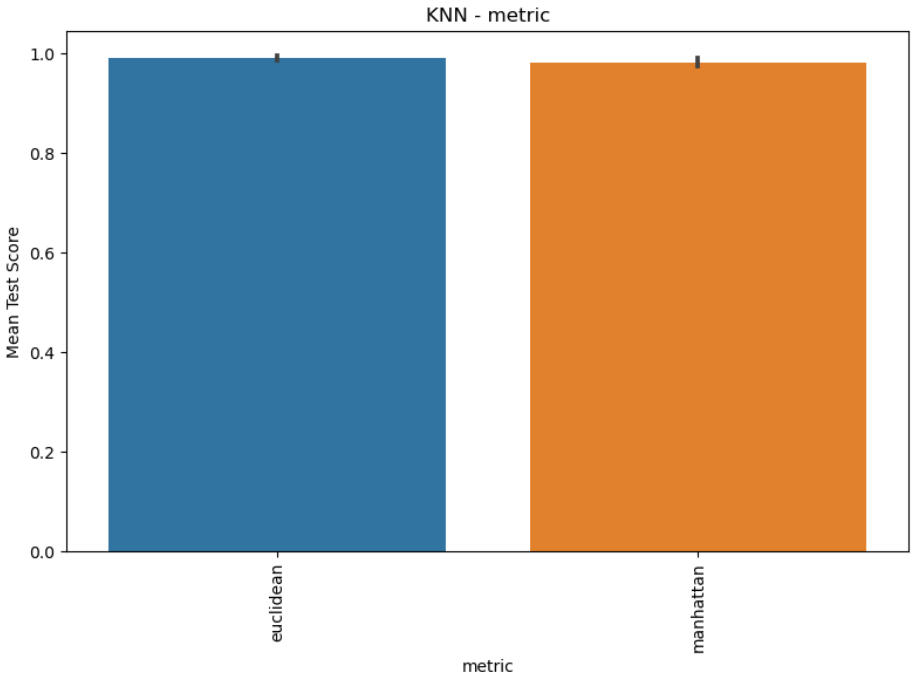


Figure 14: KNN Hyperparameter Tuning Results: metric

5.3 Naive Bayes

For Naive Bayes, the group used 3-fold grid search to tune the hyperparameters and used accuracy as the objective function. The hyperparameter grid and results were shown in **Table 3**.

Table 3: Summary of Naive Bayes hyperparameter Tuning

hyperparameter	Definition	Parameter Grid	Optimal Value
alpha	moothing parameter to handle zero probabilities	0.01, 0.12, 0.23, 0.34, 0.45, 0.56, 0.67, 0.78, 0.89, 1.0	0.01

Alpha is the smoothing parameter used in Naive Bayes to handle zero probabilities. It is perticularly important in multinomial and complement Naive Bayes, which is not the case in this report. As shown in **Figure** 15, alpha does not have a big impact on the model performance. From the grid search, 0.01 is the optimal alpha.

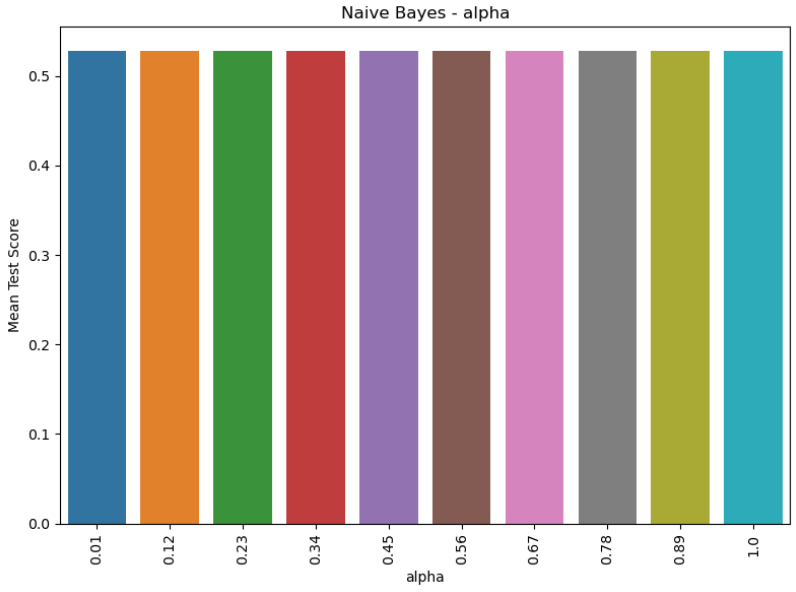


Figure 15: Naive Bayes Hyperparameter Tuning Results: alpha

5.4 Random Forest

For Random Forest, the group performed 100 runs of 3-fold CV random search to tune the hyperparameters and used accuracy as the objective function. The hyperparameter grid and results were shown in **Table 4**.

Table 4: Summary of Random Forest hyperparameter Tuning

hyperparameter	Definition	Parameter Grid	Optimal Value
n_estimators	number of trees	10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190	190
max_depth	max depth of each tree	3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19	19
min samples split	minimum number of samples required to split an internal node	2, 3, 4, 5, 6, 7, 8, 9	3
min samples leaf	minimum number of samples required to be at a leaf node	2, 3, 4, 5, 6, 7, 8, 9	2
max features	number of features to consider when looking for the best split	log_2, auto, sqrt	sqrt

Generally, more trees lead to better performance but increase computational cost. As shown in **Figure** 16, when $n\_estimator = 50$ or 160, the mean test score is the high with low variance. However, since the task is trying to find the optimal combination set of the hyperparameters and random search is search for the optimal local minimum, 190 as selected as the optimal $n\_estimator$. max_depth is the max depth of each tree. Generally, deeper trees can capture more complex patterns but may lead to overfitting. As shown in **Figure** 17, when the max_depth is 14 or higher, the mean test score almost reaches the plateau. From the random search, 19 is the optimal max_depth. $min\_samples\_split$ is minimum number of samples required to split an internal node and it is used to control the growth of the trees to help prevent overfitting. As shown in **Figure** 18, $min\_samples\_split$ does not have a big impact on the mean test score. From the random search, 3 is selected as the optimal $min\_samples\_split$. $min\_samples\_leaf$ is minimum number of samples required to be at a leaf node. Similar to $min\_samples\_split$, it is used to control the growth of the trees to help prevent overfitting. As shown in **Figure** 19, when $min\_samples\_leaf = 2$, the model has the highest mean test score with lowest variance. Thus, 2 is the optimal $min\_samples\_leaf$. $max\_features$ is number of features to consider when looking for the best split and it introduces randomness to help create diverse trees to improve model performance. As shown in **Figure** 20, sqrt is the optimal choice for $max\_features$.

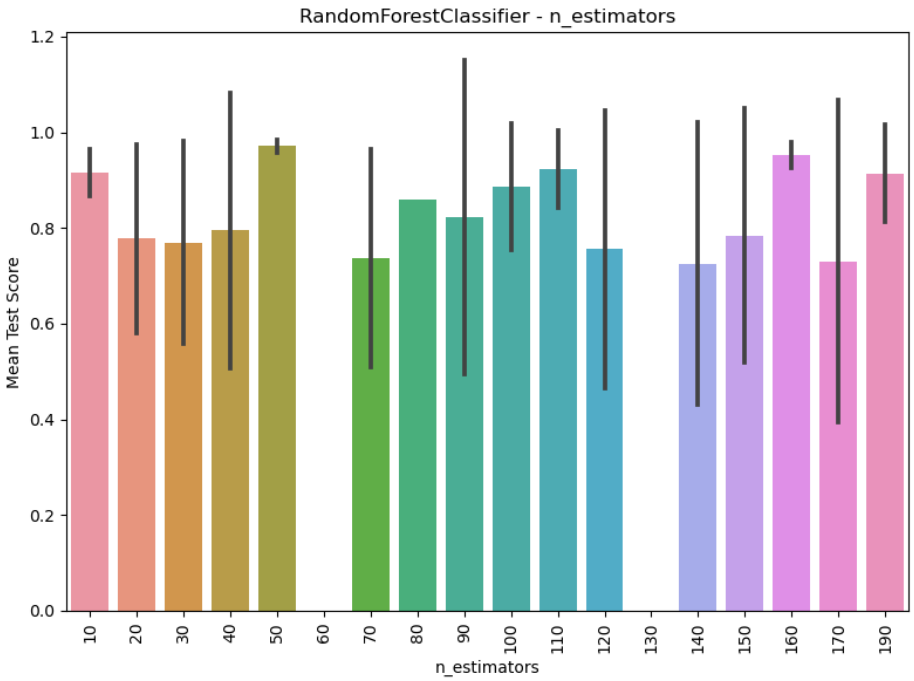


Figure 16: Random Forest Hyperparameter Tuning Results: n_estimator

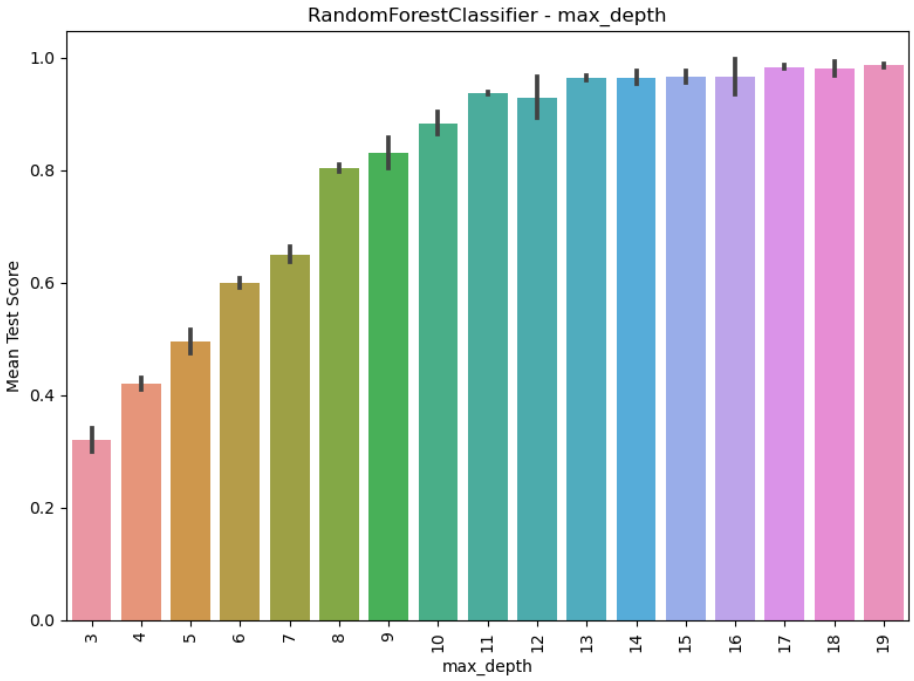


Figure 17: Random Forest Hyperparameter Tuning Results: max_depth

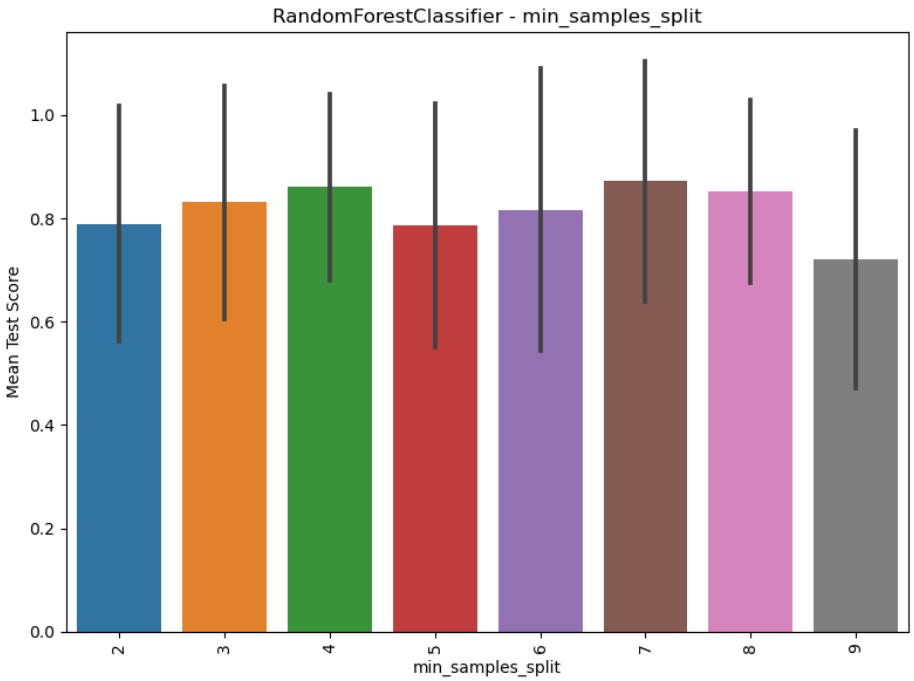


Figure 18: Random Forest Hyperparameter Tuning Results: min_sample_split

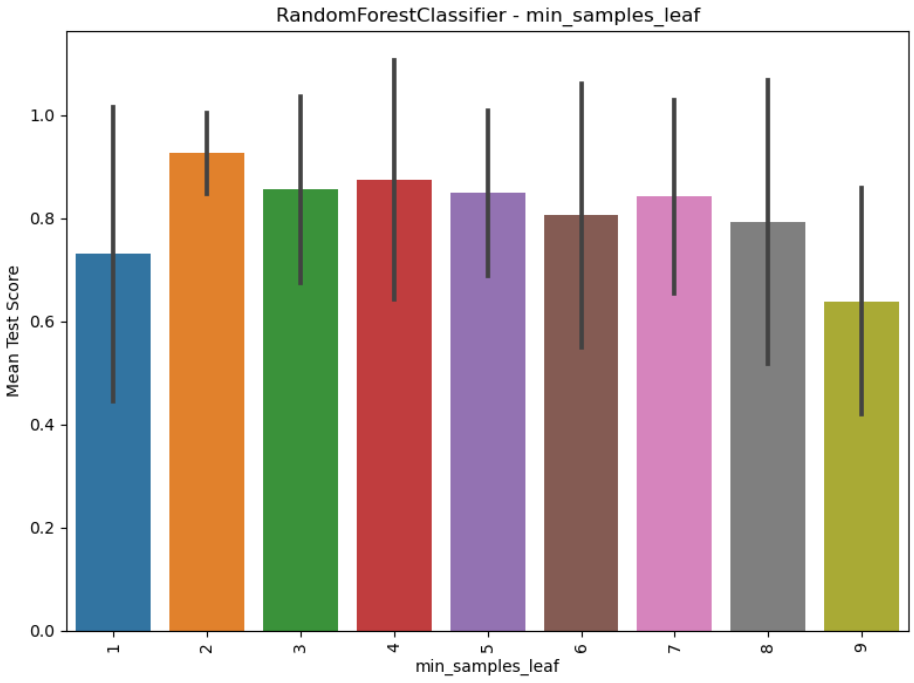


Figure 19: Random Forest Hyperparameter Tuning Results: min_sample_leaf

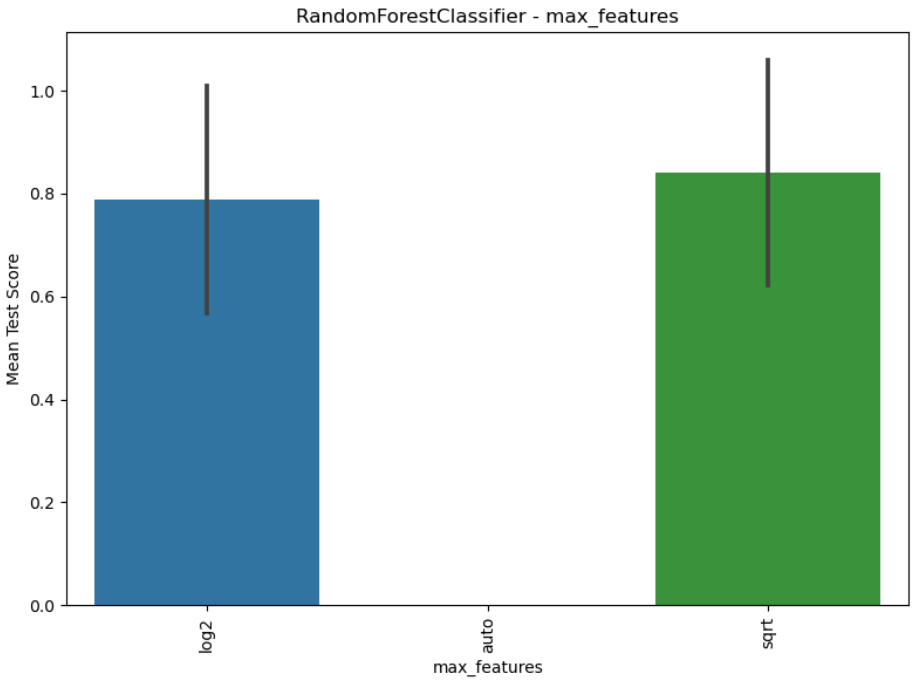


Figure 20: Random Forest Hyperparameter Tuning Results: max_features

5.5 Neural Network

Similar to other machine learning models [6], the data is transformed using StandardScaler() prior to training the MLPClassifier. The model is set to iterate up to 1000 times and is configured with a random state to ensure reproducibility of results. While various iteration values were tested, setting the iteration count to 1000 ensures robust convergence of the model.

The initial configuration of the MLPClassifier includes the following hyperparameters: the activation function is set to "relu", the solver to "adam", and the learning rate to "adaptive". The "relu" activation function is chosen for its simplicity and effectiveness in image processing tasks, while the "adam" solver is selected for its efficiency in handling large vectorized datasets from thousands of images. In addition, the adaptive learning rate is used instead of the constant method to dynamically adjust the learning rate during training, ensuring better convergence and improved performance across varying data distributions.

The default model achieves an F1 score of approximately 0.80. While the model performs well on numerical data, it is less effective on text-related hand signals. Specifically, the F1 scores for the letters K, G, and F are 0.66, 0.80, and 0.81, respectively.

To optimize the model's performance, a grid search is conducted to tune the hidden_layer_sizes and alpha parameters using GridSearchCV(). The results of this parameter tuning are presented in table below:

As observed from the results, the accuracy generally improves as the number and sizes of hidden layers increase along with the alpha value, until it reaches a peak accuracy of 0.83 with the configuration (200, 200) and alpha 0.05. Beyond this point, the accuracy slightly decreases with the configuration (250, 250) and alpha 0.06. Although computationally expensive, the grid search strategy effectively identified the optimal hyperparameters for maximizing model performance.

Therefore, the results indicate that the highest accuracy of 0.83 was achieved with a hidden layer configuration of (200, 200) and an alpha value of 0.05. These findings suggest that a moderate increase in hidden layer complexity and regularization parameter enhances the model's performance.

Table 5: Accuracy achieved for different combinations of hidden layer sizes and alpha values

hidden_layer_sizes	alpha	accuracy
(50, 50)	0.01	0.76
(50, 50)	0.02	0.78
(100, 100)	0.025	0.79
(150, 150, 150)	0.05	0.80
(150, 150)	0.03	0.81
(200, 200)	0.05	0.83
(250, 250)	0.06	0.82

6 Evaluation and Final Results

6.1 Most Computationally Efficient Model:

Table 6 summarizes the final model accuracy and hyperparameter tuning process and time cost. It shows that Neural network has the best model performance in terms of model accuracy. In terms of computational efficiency, Random Forest only took 8 min to finish 100 runs of random search, which outperformed significantly other models.

Table 6: Summary of accuracy and hyperparameter tuning of different ML models

	Accuracy (%)	**# of Run in Hyperparameter Tuning**	**Time Cost (min) in Hyperparameter Tuning**
Neural Network	83.0	7	40
Random Forest	81.7	100	8
KNN	81.5	16	17
XGboost	80.0	100	168
Naive Bayes	47.9	10	1.5

6.2 Highest Accuracy Model:

The neural network model outperformed other models by capturing subtle data patterns. Its deep structure allowed it to learn complex features that simpler models often miss. Starting with ReLU activation and the Adam optimizer worked well for the large image dataset. The model's adaptive learning rate helped it train more effectively. Fine-tuning through extensive testing of different setups led to an optimal configuration: two hidden layers of 200 neurons each and a regularization strength of 0.05. This achieved 83% accuracy, the highest among all tested models. The neural network's ability to handle the dataset's complexity ultimately gave it the edge.

7 In Summary

The group's work involved a substantial amount of data preprocessing and compute time compared to previous research projects on the topic of ASL image recognition. Through the implementation of image generation techniques and extensive hyperparameter tuning, the group achieved an accuracy of 83%. Testing the model performance without artificial image generation for class balancing resulted in significantly lower accuracy. Overall, this project provided a unique opportunity to apply the models discussed in the course and explore methods for balancing minority classes.

References

[1] American Sign Language, *American Sign Language — Commission on the Deaf and Hard of Hearing (CDHH)*, https://cdhh.ri.gov/information-referral/american-sign-language.php, n.d.

[2] R. Makwana, *American sign language recognition dataset*, Accessed: 2024-06-01, 2021. [Online]. Available: https://www.kaggle.com/datasets/rahulmakwana/american-sign-language-recognition.

[3] A. Kumar, *Hand sign images dataset*, Accessed: 2024-06-12, 2020. [Online]. Available: https://www.kaggle.com/datasets/ash2703/handsignimages.

[4] A. Raj, *American sign language dataset*, Accessed: 2024-06-12, 2021. [Online]. Available: https://www.kaggle.com/datasets/ayuraj/asl-dataset.

[5] S. Shivashankara and S. Srinath, "A review on vision based american sign language recognition, its techniques, and outcomes," in *2017 7th International Conference on Communication Systems and Network Technologies (CSNT)*, 2017, pp. 293–299. DOI: 10.1109/CSNT.2017.8418554.

[6] A. Wahane, R. Gadade, A. Hundekari, A. Khochare, and C. Sukte, "Real-time sign language recognition using deep learning techniques," in *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, 2022, pp. 1–5. DOI: 10.1109/I2CT54291.2022.9825192.