INF264
Project 2:
Digit recognizer

Alina Artemiuk

Fall 2023

I. Summary

We were approached by the Chief Elf Officer of Santa's Workshop to assist in developing a system capable of quickly and accurately identifying handwritten characters on gifts. These characters, ranging from the numbers 0-9 to the letters A-F, are crucial as they play a pivotal role in the sorting process, ensuring each gift reaches the right child. We developed a solution that demonstrated a 97.72% accuracy rate in our tests, meaning it could correctly identify and sort 97 out of 100 gifts based on their labels.

However, in the context of Christmas magic, a 3% error rate equates to some children not receiving their expected gifts - a scenario that is simply unacceptable. Although our system is promising, it still needs improvements to increase its accuracy to ensure that the holiday is as magical as it should be.

II. Technical report

1 Observations about the Dataset

The dataset, derived from the EMNIST (Extended MNIST) dataset, consists of 107,802 flattened images of size 20×20, each associated with a corresponding label. Each image depicts a handwritten hexadecimal digit and is labeled with integers or letters, ranging from 0 to 9 and A to F, respectively. An additional category is designated for empty images, resulting in 17 distinct classes in total. These grayscale images have pixel values ranging from 0 (black) to 255 (white).

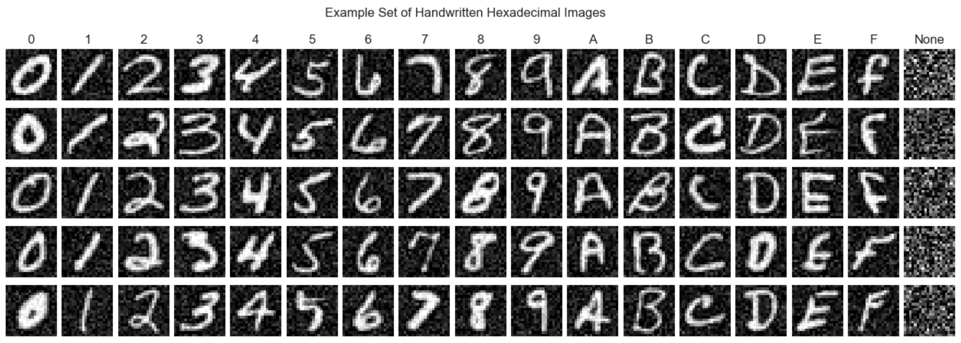

Figure 1: Sample images from each class in the dataset.

The dataset's class distribution is imbalanced, with some classes, particularly 14 (with label 'E'), having considerably fewer samples than others. This disparity can lead to potential biases in models towards overrepresented classes. In subsequent steps, methods such as class weighting and the SMOTE technique will be employed to address and mitigate this imbalance.

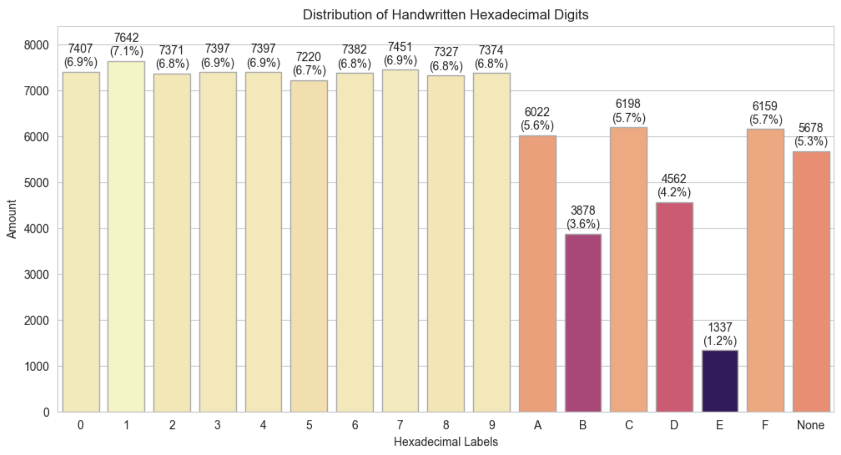

Figure 2: Distribution of classes in the dataset.

2 Pre-processing Steps

2.1 Data Exploration

Visualization tools, including matplotlib and seaborn, were used to conduct a detailed assessment of the dataset's structure and content.

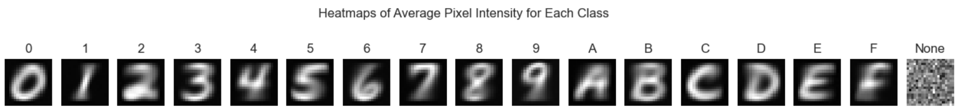

Figure 3: Heatmaps of average pixel intensity for each class.

The heatmaps of average pixel intensity for each class reveal distinct patterns. However, certain characters, notably '0' and 'D' or '8' and 'B', exhibit similarities, possibly due to their naturally similar structure, which might introduce challenges during classification.

2.2 Data Splitting

The dataset was divided into a 70:30 ratio for training and testing, respectively, ensuring a large enough training set to be later used with cross-validation techniques. Stratification guaranteed that both the training and test sets would have a similar distribution of target classes as in the original dataset. This approach helps to preserve the inherent structure of the dataset and reduces the risk of introducing biases during model evaluation.

2.3 Data Scaling

All features underwent normalization to attain a uniform [0, 1] scale. Using raw pixel values can slow down training, as many optimization algorithms operate best when inputs are within a consistent, smaller scale. By scaling to [0, 1], the training process becomes more stable and often yields better performance in image classification tasks.

2.4 Handling Data Imbalance

Initial attempts to manage data imbalance by applying the SMOTE (Synthetic Minority Over-sampling Technique) to the entire training dataset yielded unsatisfactory results, as visualized in the Figure 4.

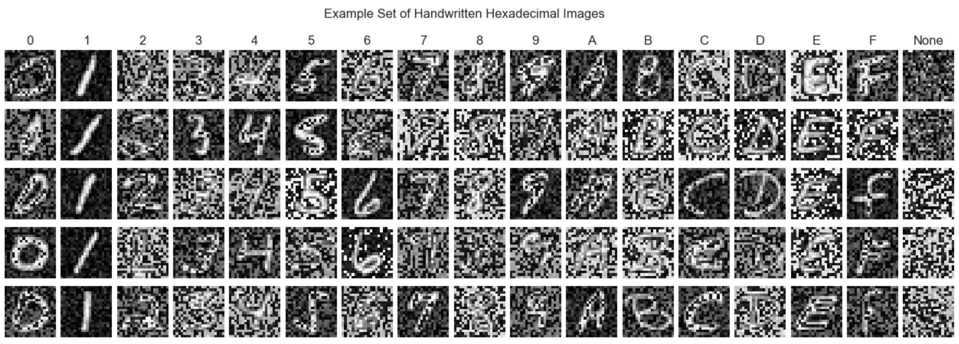

Figure 4: Results following the preliminary application of SMOTE on the training dataset.

These results are partly explained by noise in the data. Given this insight, it was decided to first denoise the dataset. Additionally, a further realization was that SMOTE should be applied during cross-validation, ensuring it's only used on the pure training data, excluding the validation subset, to prevent potential data leakage and the risk of overfitting.

It's worth noting that the decision to use SMOTE over other resampling techniques, like simple oversampling or undersampling, was driven by the significant disparity in class distribution. For instance, class 14 (with label 'E') has only 1,337 samples, whereas class 1 (with label '1') has 7,642. Simply oversampling could lead to overfitting due to duplicate entries, and undersampling could result in the loss of valuable data. SMOTE, by generating synthetic samples, offers a more balanced approach that doesn't replicate the same data or reduce the dataset size, making it a preferred choice in such scenarios.

2.5 Denoising Autoencoder

A denoising autoencoder was employed to cleanse the noisy data and improve feature extraction for the classification models. The autoencoder, consisting of encoding and decoding layers, underwent 50 epochs of training to learn optimal input data reconstruction. Both training and test data were then processed with this autoencoder, yielding denoised features ready for classifier training.

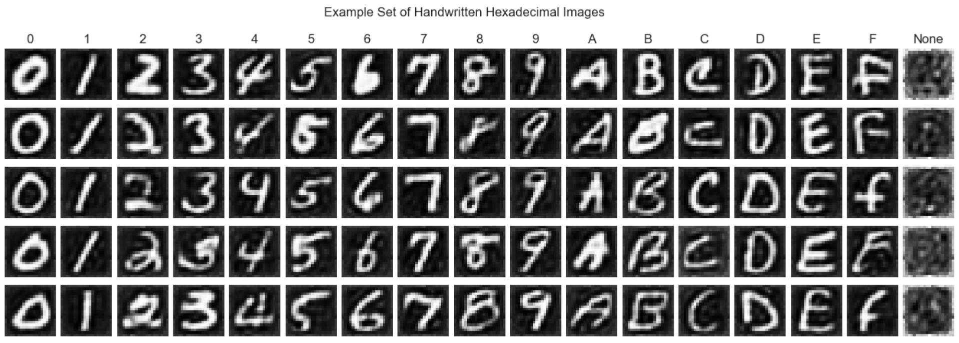

Figure 5: Post-denoising data using the autoencoder.

2.6 Dimensionality Reduction using Principal Component Analysis (PCA)

Dimensionality reduction, specifically using PCA, is a powerful technique to simplify the representation of data, reducing the number of features while retaining the majority of the data's variance or information. By capturing the data's primary directions of variance, PCA allows a comprehensive yet concise representation, often resulting in increased computational efficiency and reduced risk of overfitting during modeling.

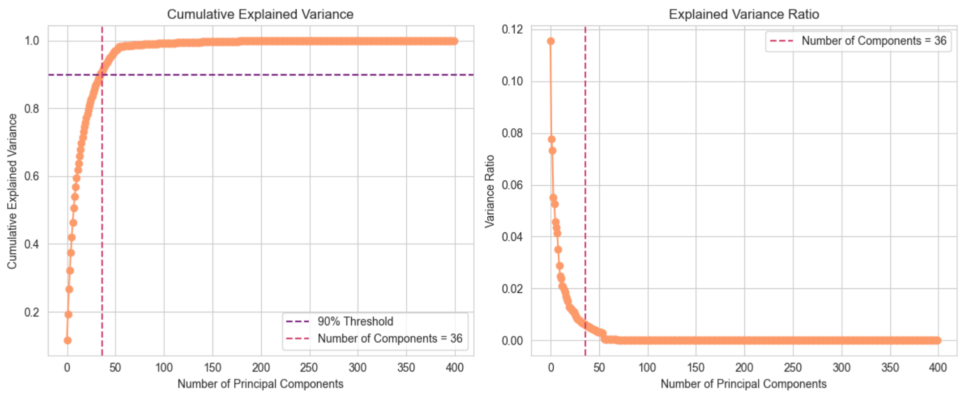

Figure 6: PCA Cumulative Explained Variance and Explained Variance Ratio.

The left plot in Figure 6 showcases the cumulative explained variance as the number of principal components increases. The purple dashed line indicates the 90% threshold, highlighting that only 36 principal components are necessary to capture 90% of the dataset's variance. Beyond this point, additional components contribute minimally to the explained variance, suggesting diminishing returns on including more components.

The Scree plot (on the right) shows the fraction of total variance explained by each component. The first few components explain a significant portion of the variance, while later components contribute less and less. Around 36 components seem to capture a significant portion of the total variance, as indicated by the "elbow" in the plot.

Given this analysis, the dataset was transformed to a 36-dimensional space using PCA, where these 36 principal components capture 90% of the total variance. This new, lower-dimensional representation is not only computationally more efficient but also retains the essential characteristics of the original data, ensuring robust modeling in subsequent steps.

3 Candidate Models and Hyperparameters

3.1 Models

1. **K-Nearest Neighbors**: KNN is valued for its simplicity and adaptability. The model works with different types of data and patterns, taking into account the k of the nearest data points during classification.

2. **Decision Trees**: These trees are appreciated for their intuitive, hierarchical decision-making structure, which makes them suitable for efficient classification of a wide array of complex features.

3. **Support Vector Machines**: SVMs are recognized for managing high-dimensional spaces and forming complex decision boundaries, which is crucial in tasks requiring detailed feature differentiation.

4. **Random Forest**: By integrating multiple decision trees, this model improves prediction accuracy and robustness, ensuring efficient handling of diverse data representations.

Other models were excluded from consideration in favour of ease of understanding and clarity. The chosen models provide a good balance between performance and interpretability.

3.2 Hyperparameters

Each model comes with specific hyperparameters that influence its learning process:

1. **K-Nearest Neighbors**:

   - `n_neighbors`: Ranges from 1 to 9 in steps of 2. It determines the number of neighbors to consider for classification, influencing model complexity and accuracy.
   - `weights`: *uniform* or *distance*. Indicates how much weight to give to each neighbor, with *distance* giving more weight to closer neighbors.

- `metric`: *euclidean* or *manhattan.* Specifies the distance metric used for calculating the distance between data points.

2. **Decision Trees**:
   - `criterion`: *gini* or *entropy.* The function to measure the quality of a split, aiding in deciding the optimal feature to split on.
   - `max_depth`: *None*, 10, or 20. Limits the maximum depth of the tree, controlling overfitting and model complexity.
   - `min_samples_split`: 2, 4, or 6. The minimum number of samples a node must contain before it can be split, influences the tree's granularity.
   - `min_samples_leaf`: 1, 2, or 4. The minimum number of samples a leaf node must have, affects the tree's depth and overfitting.

3. **Support Vector Machines**:
   - `kernel`: *linear*, *rbf*, or *poly.* Determines the kernel function to transform the input space, affecting the decision boundary.
   - *`C`*: 0.1, 1, or 10. The regularization parameter, balances the trade-off between achieving a low training error and a low validation error.
   - For *poly* kernel: `degree` can be 2, 3, or 5. Specifies the polynomial degree in the kernel, influencing model complexity.

4. **Random Forest**:
   - `criterion`: *gini* or *entropy.* The function used to evaluate the quality of a split, is similar to decision trees.
   - `n_estimators`: 100, 200, or 400. Indicates the number of trees in the forest, impacting the model's accuracy and computation time.

3.3 Justification for Hyperparameter Values Selection

When selecting values for hyperparameters, several factors come into play: theoretical understandings of the model, and practical considerations related to computation time and resources. The goal is to explore a broad but reasonable space of values to ensure we find a combination that allows the model to learn effectively from the data without overfitting.

For instance, in the case of the K-Nearest Neighbors model, odd values for k (the number of neighbors) are typically chosen to avoid ties. The range is selected to explore the trade-off between a smaller k (more sensitive to noise, but able to capture complex patterns) and a larger k (smoother decision boundaries, but may oversimplify the model).

In decision trees and random forest models, the depth of the tree and the minimum samples needed to make a split or form a leaf node are crucial. Smaller trees are computationally efficient and less likely to overfit but may be too simple to capture underlying patterns in the data. A range of values were explored to balance these considerations.

For the SVM, the C parameter controls the trade-off between having a smooth decision boundary and classifying the training points correctly. A low C creates a smoother decision surface, while a high C aims to classify all training examples correctly by giving the model freedom to select more samples as support vectors.

The choice of values is, therefore, a mix of theoretical best practices and practical considerations to ensure a comprehensive search without excessive computational requirements.

4 Hyperparameter Tuning Analysis

4.1 K-Nearest Neighbors

4.1.1 Hyperparameter Impact

1. **Number of Neighbors (`n_neighbors`)**: KNN exhibited surprisingly good performance with both small and large values of neighbors k. The model adapted well to different values of k, showcasing its flexibility.

2. **Weighting Scheme (`weights`):** The choice of weighting scheme had varying effects on model performance, depending on the specific k value. In general, the *distance* weighting scheme tended to perform slightly better when k was larger, emphasizing the influence of closer neighbors.

3. **Distance Metric (`metric`):** Both distance metrics, *euclidean* and *manhattan*, performed similarly in terms of validation score. However, on average, the *euclidean* metric demonstrated a slight advantage in performance.

4.1.2 Validation Scores

The validation scores obtained after hyperparameter tuning consistently fell within the range of approximately 94% to 95%. This performance stability suggests that the choice of hyperparameters, including the distance metric and weighting scheme, did not have a significant impact on the model's ability to achieve high accuracy in this particular task.

4.1.3 Timing

The choice of model metric had the most substantial impact on the scoring time of the K-Nearest Neighbors model, with the *manhattan* metric resulting in significantly longer scoring times compared to the *euclidean* metric. On average, the scoring time ranged from approximately 4 to 20 seconds across various hyperparameter configurations.

4.1.4 Hyperparameter Tuning Results

Hyperparameter tuning was conducted to optimize the KNN model's performance. Surprisingly, the tuning process did not lead to significant variations in the results. The validation scores across different hyperparameter configurations remained consistently high and within a relatively narrow range.

4.1.5 Plot

To visualize the hyperparameter tuning results, Figure 7 displays the performance of the KNN model under different hyperparameter configurations.

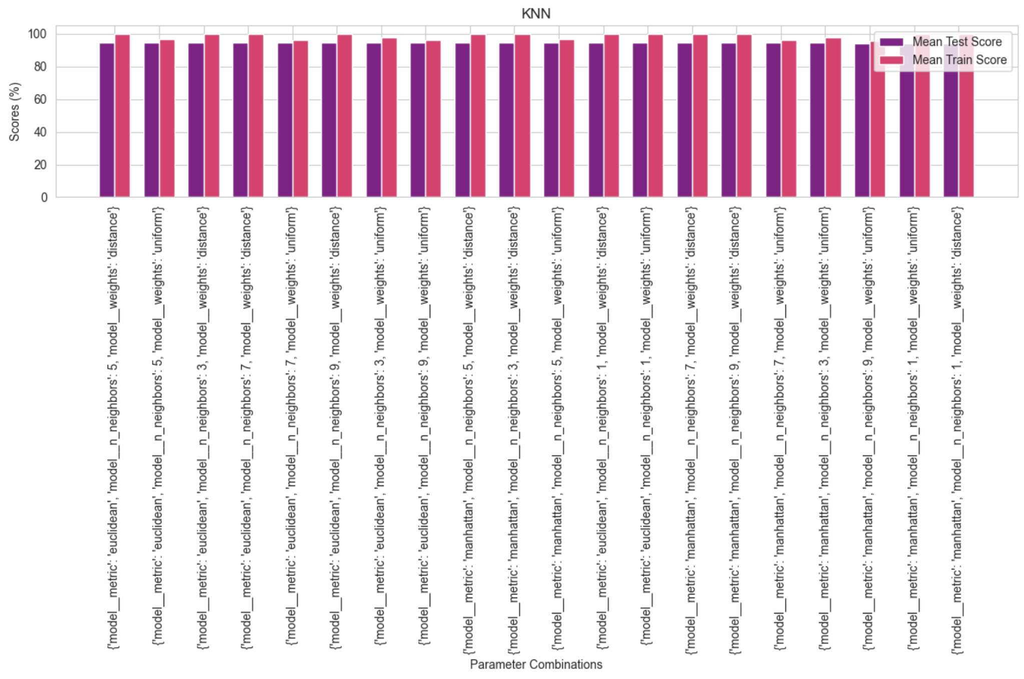

Figure 7: K-Nearest Neighbors Hyperparameter Tuning Results.

4.2 Decision Tree

4.2.1 Hyperparameter Impact

1. **Criterion (`criterion`)**: The *entropy* criterion consistently outperformed the *gini* criterion across different hyperparameter configurations.
2. **Maximum Depth (`max_depth`)**: Models with deeper trees tended to overfit the training data, as indicated by high training scores and lower validation scores. Shallow trees, on the other hand, exhibited better generalization but with some loss in training accuracy.
3. **Minimum Samples per Leaf (`min_samples_leaf`)**: Increasing the minimum number of samples required to form a leaf node generally improved model generalization. Smaller values of this hyperparameter led to overfitting, while larger values resulted in more balanced trees with better validation performance.
4. **Minimum Samples per Split (`min_samples_split`)**: Similar to the minimum samples per leaf, increasing the minimum number of samples required to perform a split improved model generalization. Smaller values of this hyperparameter led to more complex and overfit trees.

4.2.2 Validation Scores

Hyperparameter tuning led to a wide range of validation scores, with the best case achieving 77% and the worst case at 60%. These results underscore the critical role of hyperparameter selection in determining model effectiveness on the validation dataset.

4.2.3 Timing

Training times ranged from 4 to 9 seconds, with an average scoring time of approximately 3.87 ms.

4.2.4 Plot

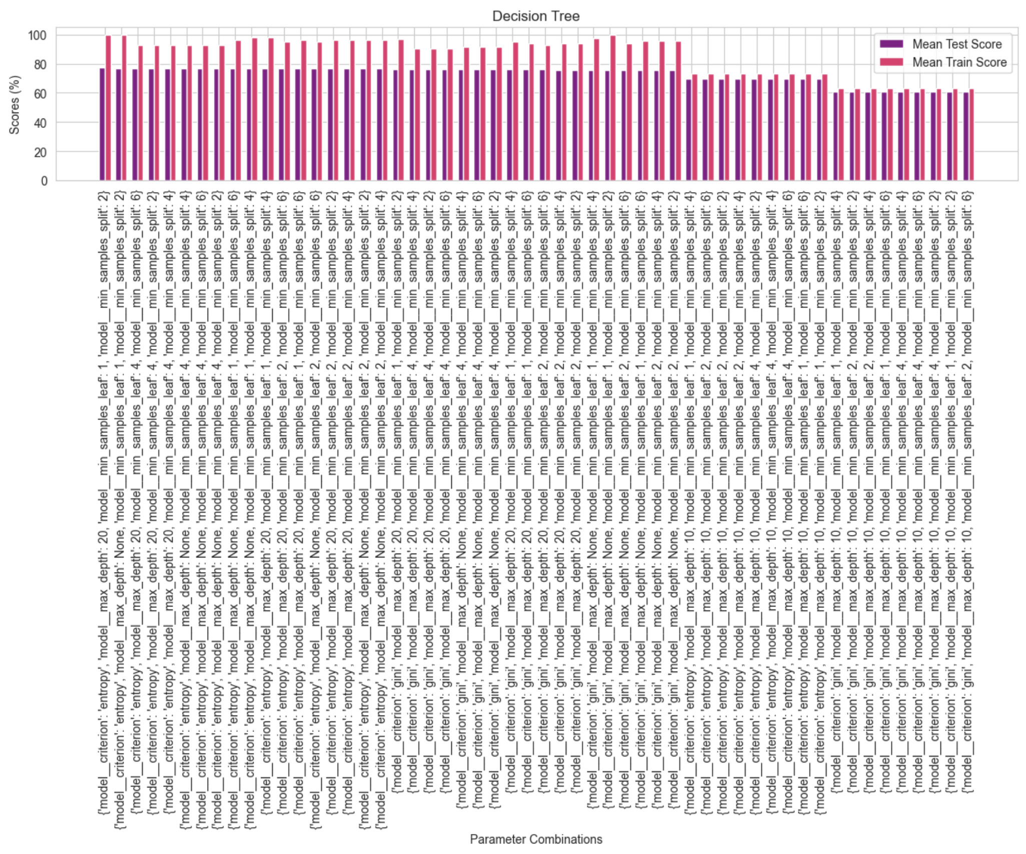

Figure 8: Decision Tree Hyperparameter Tuning Results.

4.3 Support Vector Machine

4.3.1 Hyperparameter Impact

1. **Kernel Function (`kernel`)**: The *rbf* kernel generally outperformed others, especially with a higher C value. The performance of the *poly* kernel depended on the degree of the polynomial, and the *linear* kernel gave stable, though not optimal, results.

2. **Regularization Parameter (`C`)**: The SVM model's performance was closely related to the value of the regularization parameter C. Higher values of C led to better training scores, indicative of a more complex model, while smaller values resulted in less complexity, mitigating the risk of overfitting.

3. **Degree of Polynomial Kernel (`degree`)**: The choice of degree impacts the performance of the *poly* kernel. Higher degrees led to increased model complexity and longer training times. A degree of 3 appeared to be a sweet spot, balancing performance and computational efficiency.

4.3.2 Validation Scores

The validation scores varied broadly, ranging from approximately 88% to 97%. The *rbf* kernel consistently offered high performance, especially with increased C values. The *poly* kernel's performance was heavily influenced by the degree of the polynomial and the C value, while the *linear* kernel offered consistent, but not peak, performance.

4.3.3 Timing

Training and scoring times were significantly impacted by the choice of kernel and the value of C. The *poly* kernel, especially with higher degrees, resulted in prolonged training times. In contrast, the *rbf* kernel provided a good balance between performance and computational efficiency.

4.3.4 Hyperparameter Tuning Results

The SVM model exhibited noticeable performance differences under various hyperparameter configurations. The most noteworthy results were observed with a C value of 10 and an *rbf* kernel, achieving a validation score of approximately 97%.

4.3.5 Plot

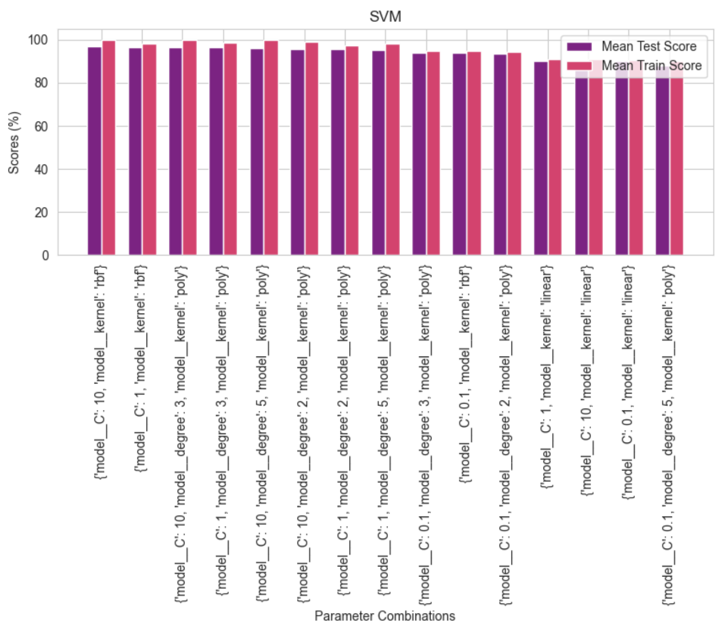

Figure 9: Support Vector Machine Hyperparameter Tuning Results.

4.4 Random Forest

4.4.1 Hyperparameter Impact

1. **Criterion (`criterion`)**: The *gini* criterion consistently outperformed *entropy* across various numbers of estimators.
2. **Number of Estimators (`estimators`)**: Increasing the number of estimators resulted in higher validation scores, although at the cost of increased training and scoring times.

4.4.2 Validation Scores

Validation scores for the Random Forest model ranged from 92.7% to 93.4%. Configurations using the *gini* criterion generally achieved higher validation scores than those using *entropy*. The number of estimators also played a significant role, with a larger number of estimators leading to slightly higher validation scores.

4.4.3 Timing

Training and scoring times increased linearly with the number of estimators. The *entropy* criterion resulted in longer training times compared to *gini*. Although higher numbers of estimators led to increased scores, the incremental benefits need to be weighed against the significantly increased computational times.

4.4.4 Hyperparameter Tuning Results

The Random Forest model achieved perfect training scores in all tested configurations, which could be indicative of overfitting. The highest validation score 93.4% was obtained with the *gini* criterion and 400 estimators. However, the increased computational time raises questions about efficiency and the potential benefits of using fewer estimators.

4.4.5 Plot

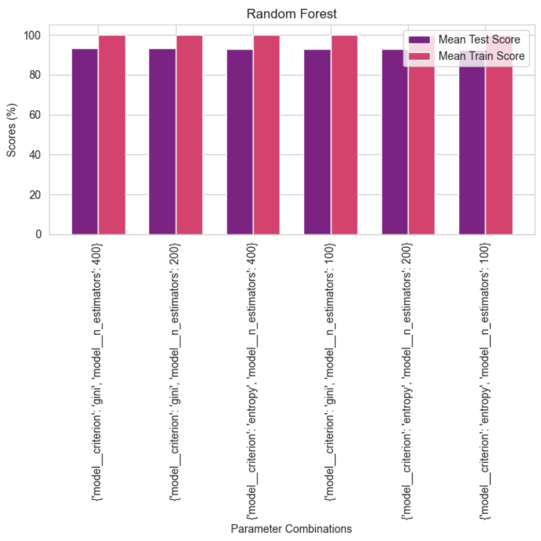

Figure 10: Random Forest Hyperparameter Tuning Results.

5 Convolutional Neural Network

At the end of the project, it was decided to implement an additional CNN model, due to its reputation for achieving outstanding performance in image recognition tasks. The objective was to find out if an increase in model complexity could enhance classification accuracy, despite the increased computational load and complexity.

5.1 Feature Extraction Before PCA

It was essential to train this CNN model before conducting PCA. CNNs are known for their ability to handle the raw image data efficiently, extracting hierarchical features that become more complex and abstract at each layer. By feeding the CNN model with raw data, we could leverage its capability to automatically learn and extract features that are more representative and discriminating for classification tasks.

5.2 Model Training

- A Convolutional Neural Network (CNN) model was built for image classification. It involved layers for feature extraction (`Conv2D` and `MaxPooling2D`) and classification (`Dense`).
- Class weights were computed to handle class imbalance, ensuring that each class was treated equally during the training.
- Early stopping was used to monitor the validation loss and stop training once the performance stopped improving, preventing overfitting.

The architecture of the CNN model is illustrated below:

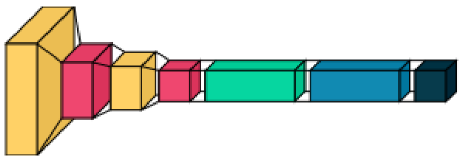

Figure 11: Visualization of the CNN model architecture.

The model architecture is taken from Keras example.

5.3 Training Process and Results

The CNN model was trained using the original feature-rich dataset to leverage the model's feature learning capabilities. The training process is visually represented in the plot below, showing the accuracy and loss progression across epochs for both training and validation data.

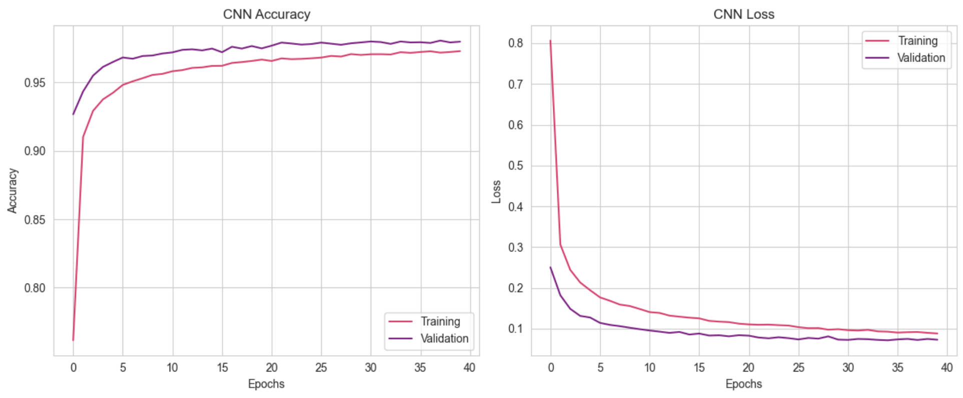

Figure 12: Training and validation accuracy and loss over epochs.

Training took approximately 149.58 seconds to complete.

6 Performance Measure

Accuracy is selected as the performance measure, attributed to its simplicity and interpretability, especially in multi-class classification scenarios like the present one involving handwritten hexadecimal digits.

In situations where datasets are imbalanced, accuracy can sometimes provide a skewed representation of model performance. However, the application of SMOTE mitigates the issues associated with using accuracy in imbalanced datasets by ensuring balanced classes, making accuracy a reliable metric. The accuracy, in this context, signifies the proportion of images that have been correctly identified and classified by the model.

7 Model Selection Scheme

The model selection was conducted using Grid Search for hyperparameter tuning. This search technique tests a predefined set of hyperparameters to find the optimal combination that maximizes model performance, based on cross-validated accuracy.

The integration of SMOTE in the grid search pipeline addresses class imbalance, aligning with previous considerations (see Section 2.4) that underscore its application strictly during each cross-validation fold to the training data. Applying it to the entire dataset, including the validation set, would mean that the model is validated on synthetic samples, leading to an overly optimistic and biased estimate of the model's performance. Integrating SMOTE in the pipeline, ensures that synthetic samples are generated anew for each training fold, preserving the integrity of the validation set and providing a more accurate and unbiased estimate of the model's generalization performance.

The use of Stratified K-Fold cross-validation ensures that each fold is a representative subset of the original dataset, maintaining the overall class distribution. It is crucial for achieving unbiased and reliable performance estimates.

This approach ensures that the model evaluation is thorough, and the derived performance metrics are reliable, leading to the selection of the most effective model with optimal hyperparameters.

8 Model Selection and Performance Estimation

8.1 Final Classifier

After rigorous model training and hyperparameter tuning, the Convolutional Neural Network (CNN) emerged as the final selected classifier. The CNN architecture was chosen for its outstanding performance in image recognition tasks and its ability to effectively capture intricate features from raw image data.

8.1.1 Justification for CNN Selection

To justify the selection of the CNN as the final classifier, the validation accuracy of various models is considered. The CNN achieved the highest validation accuracy, outperforming all other models. Here are the validation accuracies for each model:

Model	**Validation Accuracy**
CNN	98.04%
K-Nearest Neighbors	94.88%
Decision Tree	77.15%
Support Vector Machine	97.04%
Random Forest	93.40%

Table 1: Model Accuracies on Validation Data.

The CNN achieved the highest accuracy, indicating its superior ability to learn complex patterns and features in the dataset. Further, the model was evaluated on test data to estimate its real-world performance.

8.1.2 Expected Performance in Production

To estimate the expected performance of the CNN on unseen data in production, it was evaluated using the test dataset. Here are the results:

- Test Accuracy: 97.72%
- Prediction time: 1.1208 seconds

The CNN demonstrated an excellent test accuracy of 97.72%, confirming its ability to generalize well to unseen data. The relatively short prediction time of approximately 1.12 seconds is also favorable for practical applications.

8.2 Classification Report

To provide a comprehensive assessment of the model's performance, a detailed classification report was generated. This report includes precision, recall, and F1-score metrics for each class. It reveals that classes 1, 6, and 16 perform exceptionally well with high precision, recall, and F1-scores, indicating precise and reliable predictions. However, class 13 presents a comparatively lower F1-score, suggesting it is a bit more challenging to classify accurately. Overall, the model demonstrates strong performance across most classes, with the macro-averaged metrics consistently at 0.97, which highlights its robustness for classifying handwritten hexadecimal digits.

```
              precision    recall  f1-score   support

           0       0.96      0.96      0.96      2222
           1       0.99      1.00      0.99      2293
           2       0.99      0.98      0.98      2211
           3       0.98      0.98      0.98      2219
           4       0.98      0.98      0.98      2219
           5       0.98      0.98      0.98      2166
           6       0.99      0.99      0.99      2215
           7       0.98      0.99      0.98      2235
           8       0.98      0.95      0.97      2198
           9       0.98      0.98      0.98      2212
          10       0.97      0.99      0.98      1807
          11       0.95      0.94      0.95      1163
          12       0.97      0.99      0.98      1860
          13       0.93      0.94      0.93      1369
          14       0.96      0.96      0.96       401
          15       0.97      0.97      0.97      1848
          16       1.00      1.00      1.00      1703

    accuracy                           0.98     32341
   macro avg       0.97      0.97      0.97     32341
```

Figure 13: Classification Report for CNN Model.

8.3 Confusion Matrix

The confusion matrix is a valuable visualization tool that provides insights into where the model makes classification errors. Below is the confusion matrix for the CNN model:

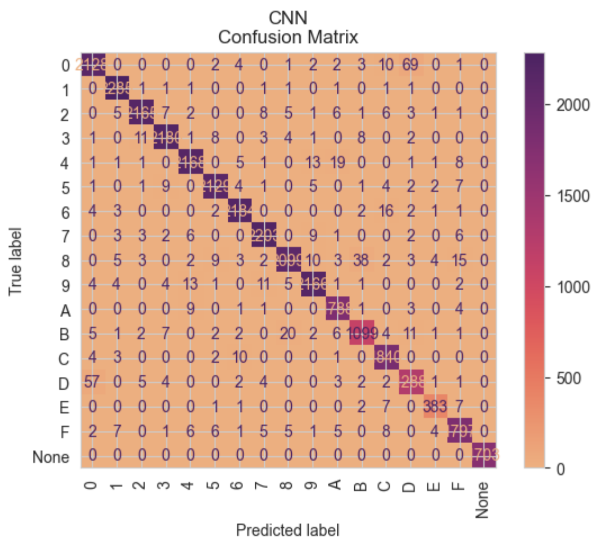

Figure 14: Confusion Matrix for the CNN Model.

The confusion matrix provides insights into the model's proficiency in classifying various classes, particularly excelling in the recognition of classes 1, 6, and 16, corresponding to labels 1, 6, and an empty image, respectively. Notably, the entire row and column associated with the empty image class (class 16) contain zeros, indicating a remarkable 100% accuracy in identifying instances where no digit is present.

However, it also reveals areas of misclassification, notably between label pairs '0' and 'D', '8' and 'B', '4' and 'A', and '4' and '9', suggesting challenges in distinguishing between these visually similar characters. This is consistent with the earlier observations regarding the classifier's potential difficulties with hexadecimal digits that exhibit similar natural structures, made during the Data Exploration (see Section 2.1).

These findings align with the F1-score analysis from the Classification Report (see Section 8.2), emphasizing the model's overall strong performance with certain classes requiring further attention to improve accuracy.

9 Overfitting Prevention

Overfitting is a significant problem in machine learning, resulting in models that are overfitted to the training data and perform poorly on unseen data. In this project, various measures were employed to mitigate overfitting:

1. **Class Weights**: Balances the classes by assigning different weights to them during the training of the CNN.
2. **Early Stopping**: Monitors validation loss during the training of the CNN and stops training when the validation loss stops improving, to prevent the model from learning the noise in the training data.
3. **Cross-Validation**: Uses Stratified K-Fold cross-validation to assess the model's performance using various subsets of the training data, ensuring a more generalized model.
4. **Principal Component Analysis (PCA)**: Reduces the dimensionality of the dataset while maintaining the essential characteristics, preventing the model from fitting to noise in the data.
5. **Grid Search with Cross-Validation**: Helps in finding the optimal hyperparameters for selected models: KNN, Decision Tree, SVM, and Random Forest, ensuring that the models do not overfit the training data.
6. **SMOTE for Imbalanced Data**: Uses the Synthetic Minority Oversampling Technique (SMOTE) to balance the class distribution, reducing the risk of overfitting on the majority class.

10 Potential Improvements

1. **GPU Support**: The models trained using Scikit-learn currently do not utilize GPU acceleration. Integrating GPU support could significantly speed up the training process, allowing for more extensive model tuning and experimentation.
2. **Enhanced Data Augmentation**: Enhancing data augmentation strategies beyond the use of SMOTE to include techniques like image rotation, scaling, translation, and applying various filters, can contribute to model robustness and improved generalization by increasing the diversity of the training dataset.
3. **Advanced Model Architectures**: Exploring complex CNN architectures could offer enhanced performance through optimized feature extraction and learning capabilities.
4. **Feature Engineering**: Investigating additional feature engineering strategies, including diverse scaling methods and the development of custom features, can help improve data representation and increase model training efficiency and prediction accuracy.
5. **In-depth Analysis of Misclassifications**: Performing detailed examinations of misclassified instances to uncover underlying patterns and characteristics can offer valuable insights, guiding refinements and optimizations to improve the accuracy and reliability of the model.