Handwritten Greek Character Recognition

Mr. Samay Rajendra Nandeshwar

Submitted for the Degree of Master of Science in

Artificial Intelligence

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

August 30, 2023

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: **13666**

Student Name: Mr. Samay Rajendra Nandeshwar

Date of Submission: 30/08/2023

Signature:

Acknowledgment

I would like to express my deepest gratitude to my dissertation guide, Dr. Nicolo Colombo, for their unwavering support and invaluable guidance throughout the journey of completing this dissertation. Their expertise, patience, and mentorship have been instrumental in shaping the direction and quality of this work.

Abstract

This dissertation explores the fascinating realm of handwritten Greek character recognition, focusing on creating effective models for accurate identification. The goal is to overcome the unique challenges presented by different writing styles, sizes, and contextual variations.

Two distinct datasets, the Greek Handwritten Characters Classification dataset and the Greek Character Database (GCDB) dataset, are used as the foundation. The datasets are carefully prepared through steps like dimensionality reduction and data organization, followed by a close look at the data's characteristics to make informed choices.

The study covers various recognition techniques, starting with basic methods like k-Nearest Neighbours (KNN) and Decision Trees, built from scratch. These serve as starting points for comparison. The Random Forest algorithm is then introduced, combining decision trees for better accuracy.

To propel recognition capabilities even further, the realm of deep learning is harnessed. Convolutional Neural Networks (CNNs) take centre stage, explored through the lenses of popular frameworks like TensorFlow. Elaborate insights into these frameworks' mechanics, coupled with effective network design strategies, are meticulously expounded.

Performance evaluation encompasses comprehensive measures, with a focus on confusion matrix-based class-wise accuracy assessment. This method encapsulates the models' performance across different classes, offering a nuanced perspective. Simulated real-world scenarios gauge the models' practical utility, while comparative analysis against existing models underscores the progress achieved.

In culmination, this dissertation not only propels the frontier of handwritten Greek character recognition but also furnishes practical understandings of varied recognition.

Contents

1 INTRODUCTION 1

2 BACKGROUND RESEARCH 3

2.1 What is the need of Greek Character recognition and general Idea. 3

2.2 **K- Nearest Neighbor 4**

2.3 **Decision Trees 5**

2.4 **Ensembled Learning and Random Forest 6**

2.5 **Convolutional Neural Network 8**

2.6 Principal Component Analysis 9

3 EXPLORATORY DATA ANALYSIS 11

3.1 Datasets 11
3.1.1 Greek Classification Dataset 11
3.1.2 GCDB Dataset 12

3.2 Preprocessing 13

3.3 Image Analysis and Clustering for Greek Classification Dataset.. 14
3.3.1 Data Loading and preprocessing 14
3.3.2 K-Means Clustering 15
3.3.3 t-SNE Visualization 15
3.3.4 Pixel Distribution Analysis 16

3.4 GCDB Dataset 17
3.4.1 Data Loading and Preprocessing 17
3.4.2 Basic Data Statistics 18
3.4.3 Sample Image Visualization 18
3.4.4 Label Distribution Plot 18
3.4.5 Applying PCA 19

3.5 Results for EDA 20
3.5.1 EDA Results for Greek Classification Dataset 20
3.5.2 EDA Results for GCDB Dataset 23

4 EXPERIMENT AND RESULT 27

4.1 Applying K-Nearest Neighbour 27

4.2 Applying Decision Trees and Random Forest 30

4.3 Applying Convolutional Neural Network 32

4.4 Results 34
4.4.1 Evaluation Of KNN 34
4.4.2 Evaluation of Decision Trees and Random Forest 39
4.4.3 Evaluation of Convolutional Neural Network. 44
4.4.4 Conclusion 47
4.4.5 Future directions 48

5 PROFESSIONAL ISSUES AND SELF-ASSESSMENT 48

6 REFERENCES 49

7 HOW TO RUN THIS PROJECT. 50

8 APPENDIX 52

8.1 Scripts 52
8.1.1 master.py Python Script 52
8.1.2 preprocessing.py Python script 53
8.1.3 EDA Greek Classification.py Python script 56
8.1.4 EDA GCDB DATAset.py Python Script 59
8.1.5 PCA.py Python Script 63
8.1.6 KNN.py Python Script 64
8.1.7 decision tress.py Python Script 70
8.1.8 Tensorflow GCDB Dataset.py python Script. 79

1 Introduction

In the realm of character recognition, Optical Character Recognition (OCR) systems have garnered significant attention due to their applications in various domains. These systems play a crucial role in converting printed or handwritten text into machine-editable formats, allowing efficient information retrieval, manipulation, and analysis. Specifically focusing on the context of Greek character recognition, a unique set of challenges arises due to variations in writing styles, sizes, and contextual complexities. The work presented by Sai Jahnavi Bachu in "Character Recognition using KNN Algorithm" [1] offers a technique for recognizing characters from noisy images using OCR. The proposed approach demonstrates the importance of enhancing image quality and utilizing classification algorithms to achieve accurate character recognition. While the mentioned paper primarily addresses English characters, the principles and methodologies it introduces can serve as a foundation for exploring similar challenges in handwritten Greek character recognition.

We employ Principal Component Analysis (PCA) as a dimensionality reduction technique to enhance the efficiency and accuracy of our recognition system. In the paper titled "Principal Component Analysis for Online Handwritten Character Recognition" [2], PCA is applied to the recognition of online handwritten characters in the Tamil script. The authors demonstrate how PCA can determine the basis vectors of class subspaces and calculate orthogonal distances for classification. Additionally, they explore techniques such as pre-clustering and modified distance measures to address limitations of traditional subspace methods. This paper serves as a guiding reference for adapting PCA to the context of Greek character recognition.

In our endeavour to enhance the accuracy of Greek character recognition, we have seamlessly integrated decision trees as a cornerstone of our methodology. This strategy is particularly adept at capturing intricate relationships within the data, aligning well with the complexities introduced by the variations in Greek characters. Moreover, in our pursuit of achieving heightened accuracies, we have harnessed the power of random forests in conjunction with decision trees. By leveraging random forests, an ensemble learning technique, we aim to further refine and amplify the predictive capabilities of our decision tree-based approach. This synergistic integration not only seeks to address the intricacies of Greek character variations but also to bolster the overall accuracy of our recognition system.

We have also utilized Convolutional Neural Networks (CNNs) to enhance the accuracy of our Greek character recognition system. In the research paper "Recognition of Handwritten Digit using Convolutional Neural Network (CNN)" by Md. Anwar Hossain and Md. Mohon Ali [3], CNNs are explored for recognizing handwritten digits with superior accuracy. This paper provides valuable insights

into how CNNs can be applied to the task of character recognition and lays the foundation for our integration of CNNs into our methodology.

In summary, this methodology for improving Greek character recognition incorporates KNN, decision trees, random forests, and Convolutional Neural Networks. By combining these techniques and drawing from existing research in character recognition and neural networks, we aim to tackle the challenges posed by Greek character variations and achieve accurate and efficient recognition results.

2 Background Research

2.1 What is the need of Greek Character recognition and general Idea.

The study and recognition of Greek characters hold a unique significance due to their historical, cultural, and linguistic importance. Greek character recognition has gained substantial attention owing to its potential to revolutionize classical studies, historical research, and linguistic analysis.

Ancient manuscripts, particularly those written in Greek, are invaluable sources of historical, philosophical, and cultural knowledge. However, deciphering these handwritten texts can be laborious and time-consuming. Handwritten Palaeographic Greek Text Recognition (HPGTR) endeavours to bridge this gap by providing researchers and classicists with digital tools to efficiently transcribe and analyse these manuscripts, enabling a deeper understanding of the past.

Manual transcription of handwritten manuscripts is time-consuming and labor-intensive. Greek character recognition through automated methods significantly accelerates the research process, allowing scholars to focus on analysing content rather than transcribing it.

The appearance of Greek characters can vary significantly based on their position in a word, their neighbours, and the font style used. This contextual variability complicates the recognition process, as character recognition systems need to account for these variations.

In a seminal study titled "Handwritten Paleographic Greek Text Recognition: A Century-Based Approach" by Paraskevi Platanou, John Pavlopoulos, and Georgios Papaioannou [4], the authors delve into the challenges of recognizing handwritten Greek paleographic manuscripts. These manuscripts exhibit characteristics that hinder accurate recognition, including multiple representations of the same character, visual similarity among characters, variations in writing style, and the influence of writing implements and practices.

By establishing the necessity of accurate Greek character recognition and drawing attention to the challenges posed by handwritten texts, we lay the groundwork for the subsequent investigations. This investigation not only addresses the complexities of recognizing Greek characters but also contributes to the broader discourse surrounding the intersection of machine learning.

2.2 **K- Nearest Neighbor**

K-nearest neighbors (KNN) is a simple and widely used classification and regression algorithm in machine learning. It is a non-parametric, instance-based learning technique that can be applied to both supervised and unsupervised tasks. [1]KNN makes predictions by identifying the "k" training examples (data points) that are closest to the input data point and then assigning a label or value based on the majority class or average value among those neighbors.

The KNN algorithm can be summarized in the following steps:

- Compute the distance between the input data point and all training data points using a chosen distance metric (e.g., Euclidean distance).
- Select the "k" training data points with the smallest distances to the input data point.
- For classification tasks, assign the class label that is most frequent among the "k" neighbors to the input data point. For regression tasks, assign the average of the target values of the "k" neighbors as the prediction.

KNN is easy to understand and implement, and it does not make strong assumptions about the underlying data distribution. However, it has limitations such as being sensitive to the choice of "k" and the distance metric, being computationally expensive for large datasets, and not performing well when the feature dimensions are not well-scaled.

The choice of distance metric is essential for determining the proximity between data points. One commonly used distance metric is the Manhattan distance, also known as the "L1 distance" or "taxicab distance." The Manhattan distance between two points is the sum of the absolute differences of their coordinates along each dimension.

[2]Let's consider two data points in a two-dimensional space: $A(x_1,y_1)$ and $B(x_2,y_2)$. The Manhattan distance between these points can be calculated using the following formula:

$$Manhattan\ Distance = /x_2 - x_1/ + /y_2 - y_1/$$

This formula can be generalized to higher dimensions as well. For A data point n dimensions $x_1, x_2, \ldots, x_n$ and another data point B with n dimensions $x_1, x_2, \ldots, x_n$, the Manhattan distance between them is:

$$Manhattan\ Distance = \sum_{i=1}^{n} |xi - yi|$$

[1] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#cite_note-1
[2] https://en.wikipedia.org/wiki/Taxicab_geometry

In KNN, the choice of the parameter "k" determines the number of nearest neighbors to consider for making predictions. A smaller value of "k" makes the model more sensitive to noise, while a larger value of "k" can lead to smoother decision boundaries but might miss finer details in the data. The distance metric used to measure the similarity between data points also plays a crucial role in the algorithm's performance.

The paper titled "Character Recognition using KNN Algorithm" by Sai Jahnavi Bachu [1] explores the application of the k-nearest neighbors (KNN) algorithm for character recognition. The KNN algorithm is employed in the context of Optical Character Recognition (OCR) to automatically detect and recognize characters from images with noise. The main steps of the proposed system include image pre-processing, segmentation, feature extraction, and classification. In the classification step, KNN is utilized to assign characters to predefined classes based on their similarity to the training dataset.

2.3 **Decision Trees**

[3]A decision tree is a widely used machine learning algorithm that is used for both classification and regression tasks. It's a graphical representation of all possible decisions and their potential consequences in a structured format resembling a tree. Each internal node of the tree represents a decision based on a certain attribute, and each leaf node represents a class label or a value. The process of constructing a decision tree involves recursively splitting the dataset based on the values of attributes until a stopping criterion is met or a leaf node with a specific class label is reached.

The decision tree algorithm works by selecting the most informative attribute (feature) to split the data at each internal node. The attribute that provides the best separation of classes or the highest reduction in impurity is chosen. The goal is to create a tree that makes accurate predictions by minimizing uncertainty or impurity at each step.

In a Decision Tree algorithm, the Gini impurity is used to measure the impurity or disorder of a dataset. The goal of the Decision Tree is to split the data in a way that reduces impurity, making the resulting subsets as pure as possible.

For a given node in the Decision Tree, the Gini impurity is calculated based on the distribution of class labels in that node. The Gini impurity *Gini(D)* of a node *D* with *c* classes is calculated using the formula:

[3] https://en.wikipedia.org/wiki/Decision_tree_learning

$$Gini(D) = 1 - \sum_{i=0}^{c} (p_i)$$

Where p_i is the proportion of samples belonging to class ii in node D.

The Decision Tree algorithm evaluates different splits of the data based on features and thresholds. For each feature and threshold combination, it calculates the Gini impurity of the resulting split subsets and selects the split that minimizes the weighted sum of Gini impurities of the child nodes.

The formula for calculating the Gini impurity of a split S is as follows:

$$Gini(S) = \frac{|S_{\text{left}}|}{|S|} \times Gini(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} \times Gini(S_{\text{right}})$$

Where:

- $|S_{left}|$ is the number of samples in the left split.
- $|S_{right}|$ is the number of samples in the right split.
- $|S|$ is the total number of samples in the split.
- $Gini(S_{left})$ is the Gini impurity of the left split.
- $Gini(S_{right})$ is the Gini impurity of the right split.

Handwritten characters can have intricate and complex relationships among their features. Decision trees work by recursively splitting the data based on single features, which might not effectively capture these complex relationships.

Decision trees are prone to overfitting, especially when they are deep and the training data is noisy or has many features. Overfitting occurs when the tree memorizes the training data instead of learning the underlying patterns. This can lead to poor generalization to new, unseen data. Complex decision trees are more likely to overfit, and controlling their depth might result in underfitting.

2.4 Ensembled Learning and Random Forest

[4]Ensemble learning is a machine learning technique that involves combining the predictions of multiple individual models (learners) to create a stronger and more accurate overall prediction. The idea behind ensemble learning is that by

[4] https://en.wikipedia.org/wiki/Ensemble_learning

combining the strengths of different models, their individual weaknesses can be mitigated, leading to improved generalization and predictive performance.

Ensemble methods are particularly useful when dealing with complex and challenging problems where a single model might struggle to provide accurate predictions.

Scikit-learn provides several ensemble methods, and two of the most popular ones are Bagging and Boosting.

[5]**Bagging (Bootstrap Aggregating):** Bagging entails training multiple copies of the same foundational model using distinct random subsets, often with replacement, from the training data. Each individual model is trained autonomously, and their predictions are consolidated to formulate the ultimate prediction. This approach aids in diminishing the model's variance, especially in scenarios involving high-variance models such as decision trees.

In this project we are using Random Forest which is an ensemble method based on the concept of bagging.

[6]The Random Forest algorithm is a powerful ensemble learning method that leverages the strength of multiple decision trees to create a more accurate and robust predictive model. It falls under the category of bagging techniques, which aims to improve the performance of individual models by combining their predictions. The primary idea behind Random Forest is to build a collection of decision trees, each trained on a different subset of the training data.

In a Random Forest, a predefined number of decision trees are constructed, with each tree being exposed to a randomly sampled subset of the training dataset. This process is known as "bagging," short for "Bootstrap Aggregating." The term "bootstrap" refers to the random sampling of data with replacement, which introduces diversity among the trees. By training each tree on a slightly different subset of data, the ensemble becomes more robust to overfitting and noise in the training data.

Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size max_features.

Individual decision trees can have a lot of variation and might overfit the data, meaning they might fit the training data too closely and not generalize well to new data. By introducing randomness in how the trees are constructed, random forests create trees that have somewhat different errors in their predictions. When you average out the predictions from these diverse trees, some of the errors can cancel each other out. This averaging helps to lower the variability in the predictions of the random forest. Although this can make the predictions slightly less accurate for individual instances, the overall reduction in variability often results in a better and more reliable model.

[5] https://scikit-learn.org/stable/modules/ensemble.html

[6] https://scikit-learn.org/stable/modules/ensemble.html

2.5 **Convolutional Neural Network**

[7]A Convolutional Neural Network (CNN) is a type of neural network that has the ability to teach itself about important aspects of data through a process called filters optimization. These filters, also known as kernels, help the network focus on specific features within the data. This innovation addresses challenges like vanishing gradients and exploding gradients, which were problematic in earlier neural networks. By employing regulated weights over fewer connections, CNNs mitigate these issues effectively.

Traditional feed-forward neural networks often exhibit full connectivity, where every neuron in one layer is connected to all the neurons in the next layer. However, this can lead to overfitting, where the network learns too much from the training data and struggles to generalize to new data. To overcome this, regularization techniques are utilized, such as weight decay or trimming connectivity. Moreover, CNNs are well-suited for diverse and robust datasets, as they are more likely to capture the underlying patterns of the dataset rather than being influenced by biases from less representative data.

A typical Convolutional Neural Network comprises an input layer, hidden layers, and an output layer. In these networks, the hidden layers encompass one or more layers that carry out convolutions. This process often involves a dot product calculation between the convolution kernel and the input matrix of the layer. An activation function like ReLU is applied to this product. As the kernel slides across the input matrix, the convolution operation generates what is called a feature map. This map then contributes to the input of the subsequent layer. The network further involves pooling layers, fully connected layers, and normalization layers, working cohesively to facilitate learning and prediction.

In a Convolutional Neural Network (CNN), the **Convolutional layers** are responsible for detecting various features in the input data through convolution operations with learnable filters. These layers capture patterns like edges, textures, and shapes, essential for recognizing complex structures within images.

Pooling layers downsample the spatial dimensions of the feature maps, reducing computational load and making the network more robust against variations in object location and size. Popular pooling techniques include max pooling and average pooling.

Fully connected layers are traditional neural network layers where each neuron is connected to every neuron in the previous and subsequent layers. They consolidate the learned features and enable the network to make final predictions. However, they increase the number of parameters significantly.

[7] https://en.wikipedia.org/wiki/Convolutional_neural_network

Receptive field refers to the region of the input that a neuron in a given layer "sees" through its connections. As we move deeper into the network, neurons perceive a wider context, enhancing the network's understanding of complex relationships.

Weights are the learnable parameters that adjust during training to minimize the difference between predicted and actual outcomes. In CNNs, weights represent the importance of different features in making accurate predictions.

In the research paper, Md. Anwar Hossain & Md. Mohon Ali, "Recognition of Handwritten Digit using Convolutional Neural Network (CNN),"[3], the authors propose a model for recognizing handwritten digits using Convolutional Neural Networks (CNN). The goal is to create a model that can accurately identify and determine the handwritten digit from its image. They use the MNIST dataset, which is a widely used in the field of machine learning for handwritten digit recognition.

The paper demonstrates handwritten digit recognition using Convolutional Neural Networks (CNNs):

- **Data**: Uses MNIST dataset of 28x28 pixel handwritten digits.
- **Model**: CNN architecture with convolutional, ReLu activation, max pooling, and fully connected layers.
- **Training**: Employ MatConvNet and SGD optimization to learn digit patterns.
- **Evaluation**: Measures accuracy on a separate test dataset.
- **Results**: CNN outperforms traditional methods, excelling in computer vision tasks.

2.6 Principal Component Analysis

[8]Principal Component Analysis (PCA) is a valuable technique employed in data analysis, particularly when dealing with extensive datasets that encompass numerous dimensions or features per observation. Its primary purpose is to reduce the complexity of data while retaining crucial information, thereby enabling better interpretation and visualization of multidimensional data.

PCA accomplishes dimensionality reduction by transforming the original dataset into a novel coordinate system, wherein the majority of data variation can be described using fewer dimensions than the initial set.

The fundamental idea behind PCA involves identifying the principal components of a set of data points in a real coordinate space. These principal components are a series of unit vectors, each denoting the direction of a line that optimally fits the data while remaining orthogonal to the preceding vectors. A line is considered the best fit when it minimizes the average squared perpendicular

[8] https://en.wikipedia.org/wiki/Principal_component_analysis

distance from the data points to the line. These orthogonal directions together constitute a new basis, characterized by linearly uncorrelated dimensions.

PCA is designed to capture as much variance as possible from the original dataset. When performing PCA, the goal is to identify the directions (principal components) along which the data varies the most. These principal components are ordered in a way that the first component captures the most variance, the second component captures the second most, and so on.

The selection of principal components is often based on how much variance they explain. Retaining the principal components that collectively explain a substantial portion of the total variance allows for a meaningful reduction in dimensionality without losing critical information.

In the research paper titled "Principal Component Analysis for Online Handwritten Character Recognition," [5] the authors describe how Principal Component Analysis (PCA) is applied to the problem of online handwritten character recognition, specifically focusing on the Tamil script. We can try and replicate the same for Greek language.

3 Exploratory Data Analysis

3.1 Datasets

There are mainly two datasets which are being used to perform this experimentation. We are calling the first dataset Greek Classification dataset [6] and we are calling the second GCDB Dataset [7] .

Both the datasets are unique in their own way, and differ a lot from each other. The Greek Classification datasets provides images in low resolution and high resolution while on the other hand the GCDB dataset provide images in quite low resolution.

The datasets differ in quantity as well, Greek Classification Datasets has only 900 images of handwritten Greek characters approximately, while on the other hand the GCDB Datasets as nearly 450 images of each Greek characters making the dataset quite large consisting of nearly 10,000 images approximately for 24 Greek characters.

I have downloaded this dataset from the link provided in the references and uploaded on my GitHub to create transparency so that a wider audience can verify and reproduce my results, contributing to the credibility of the work done.

The characteristics of the data sets is described below:

3.1.1 Greek Classification Dataset

[9]This dataset was intended to be used for experimenting with various machine learning techniques for classification purposes. Unlike the common handwritten digit's dataset, this one has more categories (24), making it a tad trickier. Creating it was enjoyable, and I'm hopeful that the data science community will find it beneficial.

The training set includes 240 pictures of Greek letters, with 10 images for each letter. The testing set contains 96 pictures, with 4 images for each letter. The dataset has offered both the original high-resolution grayscale images in two zip files (for training and testing) and the low-resolution images (14x14 pixels) used for classification in two additional zip files (for training and testing). Lastly, matrices corresponding to each low-resolution (14x14) image are provided in two .csv files (for training and testing), where the cell numbers indicate grayscale intensities.

The last column in both .csv files contains the labels/ ground truth, that is the 24 different letters. Each letter corresponds to a number in a way that it is explained below:

[9]https://www.kaggle.com/datasets/katianakontolati/classification-of-handwritten-greek-letters

Letter Symbols => Letter Labels
α=>1, β=>2, γ=>3, δ=>4, ε=>5, ζ=>6, η=>7, θ=>8, ι=>9, κ=>10,
λ=>11, μ=>12, ν=>13, ξ=>14, ο=>15, π=>16, ϱ=>17, σ=>18, τ=>19, υ=>20,
φ=>21, χ=>22, ψ=>23, ω=>24

In summary we have:

- *The original high-resolution images:* (train _high _resolution.zip, test _high _resolution.zip)
- *The low-resolution (14x14) images:* (train _letters _images.zip, test _letters _images.zip)
- *Training dataset:* Grayscale intensities- with 240 rows/data, 196 columns/features, column 197 contains the labels (train.csv)
- *Test dataset:* Grayscale intensities- with 96 rows/data, 196 columns/features, column 197 contains the labels (test.csv)

3.1.2 GCDB Dataset

[10]The GCDB (Greek Character Database) is a comprehensive system designed for the storage and management of unconstrained handwritten Greek characters in image form. This database consists of three integral components: a specialized input form, a dedicated database housing the images captured from these forms, and software tools enabling the extraction and retrieval of data from the forms into the database. The central objective of the GCDB system is to facilitate the automated organization and archival of Greek symbols and characters in image format, with the intention of supporting applications like Optical Character Recognition (OCR) systems and other relevant use cases.

The design of the GCDB system is strategically planned to accommodate future growth and expansion. It features mechanisms for effortless and rapid data insertion, ensuring that the system can be kept up-to-date with the latest collection of Greek handwritten characters. This expansion-oriented approach ensures that the database can effectively meet the rising requirements for offline character recognition of the Greek language.

As mentioned earlier this dataset consists of Greek character images, 450 images approximately, for Both Capital and Small caps Greek letters. For this project we are using small Greek character images to maintain transparency.

In summary we have 24 folders of Greek Characters each folder consists of images for individual characters, making it a quite large dataset.

Further analysis on both the datasets are mentioned onwards.

[10] https://www.kaggle.com/datasets/vrushalipatel/handwritten-greek-characters-from-gcdb

3.2 Preprocessing

As I am working with image data to prepare it for further analysis, the primary goal is to organize and process images of Greek characters, which are crucial to the research. The following steps highlight the key actions performed in the process.

- **Character Label Mapping:** I define a mapping that associates each Greek character's name with a numerical label. This mapping will help categorize the characters during the processing.

- **Cloning Git Repositories**: As I mentioned earlier that I have uploaded the Datasets on git to create transparency, The code handles the cloning of two different Git repositories. These repositories likely contain image datasets relevant to my research on Greek characters.

- **Image Processing:** The core of the process is dedicated to processing the image data. Within each repository, I loop through folders representing character categories. For each character, I select a subset of images for processing.

- **Grayscale Conversion**: Images are converted to grayscale using the convert method, reducing colour information to a single intensity value per pixel.

- **Resizing and Flattening:** Images are resized to a consistent dimension (14x12 pixels) to standardize the input. They are then flattened into a 1D array of pixel values.

- **Label Association:** The numerical label corresponding to the character category is added to the end of the pixel values array.

- **Data Collection:** The processed image data, along with associated labels, is collected into a structured format using the data list.

- **Dataframe Creation and CSV Export**: The collected data is converted into a panda's DataFrame. This DataFrame is then saved as a CSV file, preserving the processed information for further analysis.

- **Repository Cleanup:** The process ensures that the cloned repositories are properly cleaned up by removing unnecessary files, particularly within the .git directories, as in the presence of this directory the code cannot execute..

3.3 Image Analysis and Clustering for Greek Classification Dataset

As this dataset is relatively small, we are going to check its performance against the GCDB Dataset mainly on ML models which is KNN and Decision trees and Random Forest.

Before that lets try to find out the underlying patterns in the data. We are assuming that we do not have any csv file with the labels to perform the analysis further.

3.3.1 Data Loading and preprocessing

In the data loading and preparation phase, we start by importing the necessary libraries and setting the paths to the image folders. This step allows us to access the image data required for our analysis. The primary goal of this section is to load and prepare the image data for subsequent analysis. We achieve this by iterating through the images in the training folder. For each image, we load it using the Python Imaging Library (PIL) and resize it to a consistent target size. By converting the resized images into NumPy arrays, we transform them into a suitable format for further processing.

Below is the pseudo code for reference purpose:

```
# Set paths to image folders
train_folder = 'Path_to_train_folder'
test_folder = 'Path_to_test_folder'

# Define target size for image resizing
target_size = (100, 100)

# Initialize an empty list to store image data
images = []

# Load and prepare image data
for each image in train_folder:
    if image ends with '.jpg':
        # Load and resize the image to target size
        image = Load image using PIL library
        resized_image = Resize image to target size
        # Convert image to NumPy array
        image_array = Convert resized_image to NumPy array
        # Append the image array to the 'images' list
        Append image_array to images list

# Convert the list of image arrays to a NumPy array
Convert images list to flattened NumPy array
```

3.3.2 K-Means Clustering

K-Means Clustering is a widely used unsupervised machine learning technique that aims to group similar data points into clusters. As I have mentioned earlier that we are treating the dataset as if it does not have any labels. In this section, we apply K-Means clustering to our flattened image data. The motivation behind K-Means clustering is to identify inherent patterns and structures within the data. By grouping similar images into clusters, we can gain insights into the visual similarities and differences among the Greek characters. This step is crucial for preliminary exploratory analysis and can provide a foundation for subsequent visualization techniques.

Below is the pseudo code for reference purpose:

```
# Apply K-means clustering
num_clusters = 24
kmeans = Initialize KMeans with num_clusters and random state
clusters = Perform K-means clustering on flattened images
Store cluster assignments in 'clusters'

# Visualize clusters using sample images
sample_size = 5

for each cluster in first 5 clusters:
    # Get indices of images in the current cluster
    cluster_indices = Get indices of images with cluster
assignment
    # Select 'sample_size' random indices from
cluster_indices
    sample_indices = Randomly select sample_size indices from
cluster_indices
    Display sample images with titles indicating cluster
number
```

3.3.3 t-SNE Visualization

t-SNE (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique used for visualizing high-dimensional data in a lower-dimensional space. Here, we apply t-SNE to the flattened image data to reduce its dimensions while preserving the pairwise distances between data points. The purpose of this step is to create a scatter plot that visually represents the clusters identified through K-Means clustering. By projecting the data into a two-dimensional space, we can gain insights into the arrangement of clusters and potentially discover underlying structures that might not be evident in higher dimensions.

Below is the pseudo code for reference purpose:

```
# Perform t-SNE for dimensionality reduction and
visualization
tsne = Initialize TSNE with 2 components and a random state
tsne_result = Apply t-SNE to flattened images to obtain
tsne_result

# Create a scatter plot using tsne_result with cluster
coloring
Create a scatter plot using tsne_result
Color points in the plot according to cluster assignments
Add legend to indicate cluster numbers
```

3.3.4 Pixel Distribution Analysis

Pixel distribution analysis involves investigating the statistical characteristics of pixel values within the images. The statistics we explore include mean, standard deviation, variance, and median pixel values. These measures provide valuable insights into the intensity and variation of pixel values across different images. By creating histograms and overlaying normal distribution curves, we can visually assess the distribution of pixel values for each statistic. This step aids in understanding the variation in image intensities and identifying potential trends or anomalies within the dataset.

Below is the pseudo code for reference purpose:

```
# Analyze pixel distribution statistics
Calculate mean, standard deviation, variance, and median
pixel values
For each statistic:
    # Create a histogram of pixel values
    Create histogram of pixel values using Matplotlib
    # Overlay a normal distribution curve on the histogram
    Calculate normal distribution curve using mean and
standard deviation
    Overlay the curve on the histogram

# Display all histograms in a 2x2 grid
Show the plots for pixel distribution analysis
```

3.4 GCDB Dataset

The GCDB Data set is quite large as compared to the Greek Classification dataset, but the image quality is not as good as the prior one.

3.4.1 Data Loading and Preprocessing

In this section, we are loading and preprocessing the dataset of Greek characters. The images of characters are loaded, resized to a common size, flattened to create feature vectors, and labelled. This step is essential to convert raw image data into a format that can be used for further analysis and modelling. Preprocessing ensures that all images are of the same size, facilitating consistent feature extraction.

It iterates through subfolders, each representing a character, and processes the images within. The images are resized to a common size of 64x64 pixels and flattened into a 1D array. These flattened arrays are collected into the 'data' list, while the corresponding labels are collected into the 'labels' list. The data is organized into a Dataframe for further analysis.

Below is the pseudo code for reference purpose:

```
data = []
labels = []
label_mapping = {}
data_folder = 'Path to data folder'

FOR each subfolder in sorted list of subfolders in
data_folder:
    label = index of subfolder + 1
    char_folder_path = join data_folder with subfolder
    IF char_folder_path is a directory:
        map subfolder name to label in label_mapping
        FOR each image_file in sorted list of files in
char_folder_path:
            image_path = join char_folder_path with
image_file
            Load image from image_path
            Resize image to 64x64 pixels
            Flatten image into 1D array and add to data
            Append label to labels

Convert data and labels to numpy arrays

Create DataFrame 'df' from data with column names as pixel
indices
Add 'Label' column to df

Print basic statistics about df
```

3.4.2 Basic Data Statistics

Basic data statistics help us understand the characteristics of the dataset. Printing the total number of images, the number of unique labels, and the label mapping provides an overview of the dataset's size and diversity. This information is useful for making informed decisions about how to proceed with analysis and modelling.

Below is the pseudo code for reference purpose:

```
Print total images in df
Print number of unique labels in df
Print label mapping in df
```

3.4.3 Sample Image Visualization

Visualizing sample images from each label gives us an intuitive understanding of the dataset's contents. This step allows us to observe the variety and intricacies of characters present in the dataset. It also aids in identifying any potential data quality issues, such as noise or anomalies in the images.

Below is the pseudo code for reference purpose:

```
Create a plot with size 12x8
FOR each label index in range of label_mapping:
    label = label index + 1
    sample_row = select row from df where 'Label' column
equals label
    Convert sample_row values to 2D image and reshape
    Create a subplot in the plot grid
    Display sample_image using grayscale colormap
    Set subplot title as label and character name
    Disable subplot axis
Adjust subplot layout and display plot
```

3.4.4 Label Distribution Plot

Creating a plot of the label distribution provides insights into the class balance of the dataset. This information is crucial for understanding potential class imbalances that might affect model performance. It can also guide decisions on data splitting for training and testing.

Below is the pseudo code for reference purpose:

```
Create a plot with size 8x5
Create a countplot for 'Label' using data from df
Set plot title, x-axis label, and y-axis label
Display plot
```

3.4.5 Applying PCA

Principal Component Analysis (PCA) is a dimensionality reduction technique used to capture essential information from high-dimensional data. By applying PCA, we aim to reduce the dimensionality of the feature space while preserving as much of the variance as possible. This benefits us in several ways:

- **Feature Reduction:** High-dimensional data can be computationally expensive to process. Reducing the number of features while retaining meaningful information speeds up subsequent analyses.
- **Visualization:** PCA can help visualize data in a lower-dimensional space. The scatter plots and correlations between principal components provide insights into how the data clusters and how features are related.
- **Feature Interpretation:** Principal components can be interpreted to understand which original features contribute the most to their creation. This can provide insights into the characteristics of the data.
- **Noise Reduction:** High-dimensional data might contain noise or irrelevant features. PCA can help suppress noise by focusing on the principal components with the highest variance.
- **Modelling:** Reduced-dimension data from PCA can be used as input for machine learning models, potentially improving model efficiency and reducing overfitting.

Below is the pseudo code for reference purpose:

```
Define num_components as 50

Create PCA object with num_components
Create StandardScaler object
Standardize data by fitting and transforming df data
(excluding 'Label' column)
Create df_pca DataFrame from PCA-transformed data with column
names as 'PC_i'

Calculate and store basic statistics for each principal
component in pca_stats
```

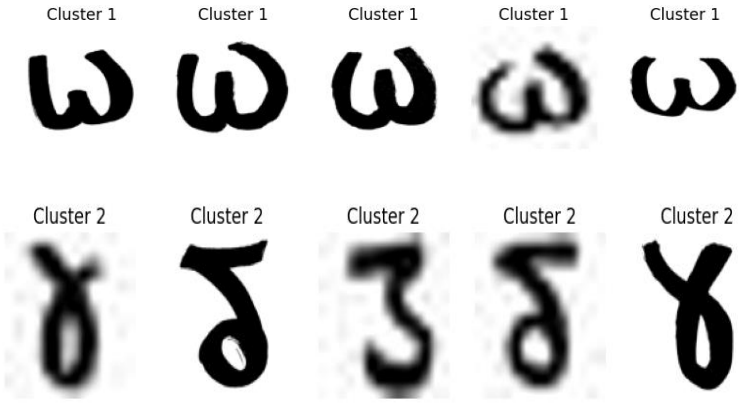
```
Plot cumulative explained variance using
pca.explained_variance_ratio_

Create scatter plot matrix of the first four principal
components using pairplot

Define num_pc_to_correlate as 10
Calculate correlations between the first num_pc_to_correlate
principal components
Print correlation matrix for the first num_pc_to_correlate
principal components

Calculate correlations between the first 10 principal
components
Create heatmap of the correlation matrix using sns.heatmap
```

3.5 Results for EDA

3.5.1 EDA Results for Greek Classification Dataset

1. Visualising K-means Clusters

After applying K-Means Clustering to the flattened image data, we obtain clusters of visually similar Greek characters and results are shown in Fig 1. As shown in the below figure each cluster represents a group of characters that share similar visual characteristics, such as shape, stroke patterns, or writing style. These clusters serve as a preliminary categorization of Greek characters based on their visual attributes. By analysing these images within each cluster, we can observe that characters within the same cluster exhibit similar visual traits. This information is essential for character recognition systems, as it enables the system to recognize variations of a particular character across different writing styles.



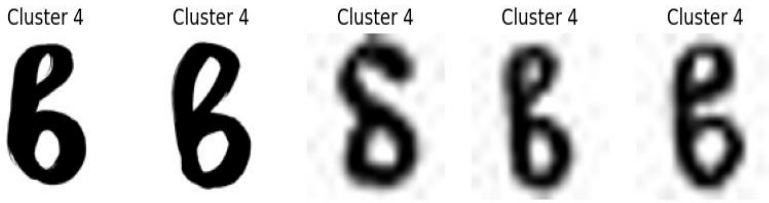


Fig 1: Sample clusters After Applying K-means

2. t-SNE Visualisation

As shown in Fig 2: The t-SNE visualization provides a two-dimensional representation of the high-dimensional image data, capturing the relationships and similarities between different Greek characters. The scatter plot obtained through t-SNE reveals the spatial distribution of clusters identified by K-Means Clustering. Clusters that appear close together in the t-SNE plot correspond to characters with similar visual patterns. Conversely, clusters that are farther apart indicate greater visual dissimilarity. This visualization allows us to intuitively understand how the clustering algorithm groups characters and how distinct different characters are

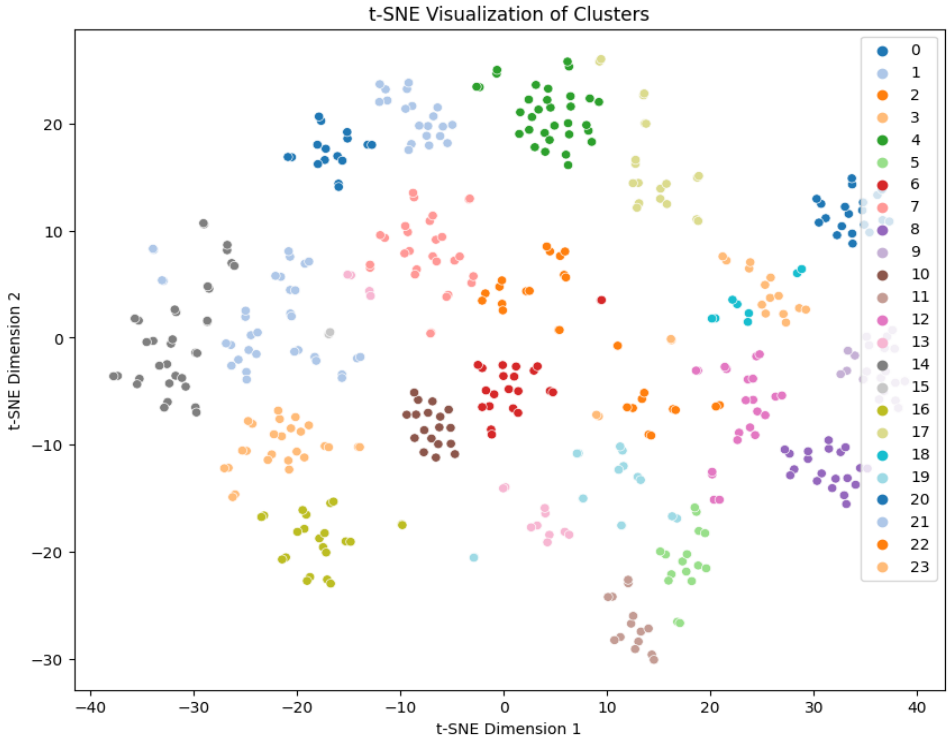


Fig 2: t-SNE Visualization of clusters for Greek Classification Dataset

from each other. It also highlights potential overlaps or ambiguities in character representations that might require further analysis.

3. Pixel Distribution Analysis

Fig 3. Pixel distribution analysis (PDA) offers insights into the intensity variations and statistical properties of pixel values within the images. Mean, standard deviation, variance, and median pixel values provide measures of central tendency and dispersion in pixel intensities. By examining the histograms and normal distribution curves for each statistic, we can observe the distribution patterns of pixel values across the dataset.

Deviations from normal-like distributions might indicate variations in character writing styles, image quality, or specific character attributes. Understanding these distribution patterns assists in detecting potential outliers,

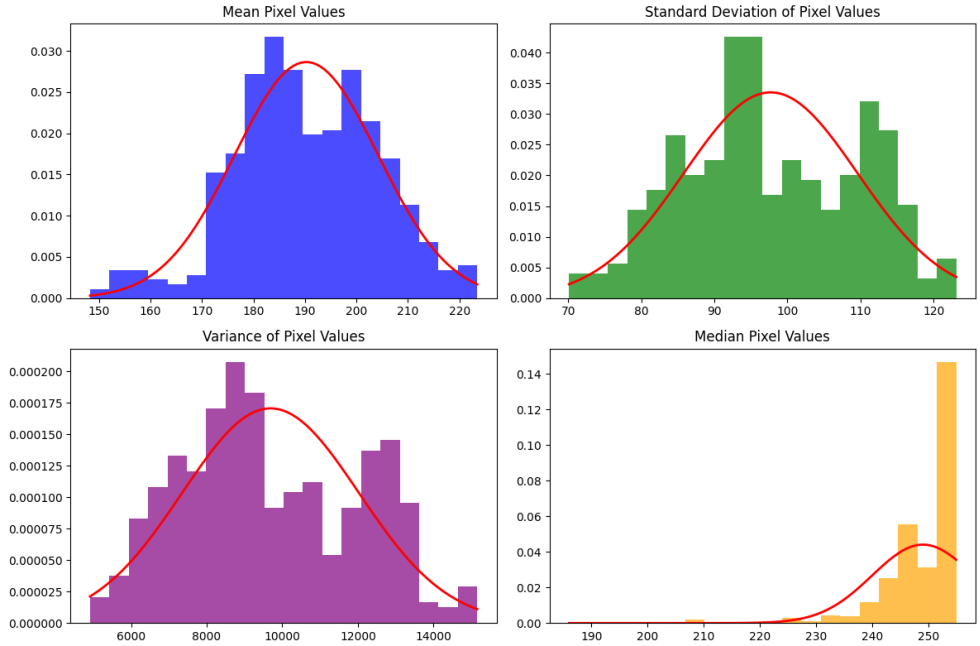


Fig 3. Pixel distribution analysis (PDA) for Greek Classification Dataset

understanding pixel value dynamics, and identifying patterns that might affect character recognition performance.

Observing at the plot we can see that the median pixel values have some variations across the dataset. There might be specific features or patterns in the images that lead to these variations. If there are significant changes in the median pixel values, it might indicate changes in writing styles, curvature, contrast, or other image characteristics.

A higher standard deviation indicates that there is a greater variability in pixel values, indicating that there can be regions of high contrast or sharp transitions between different areas in the images. On the other hand, a lower standard deviation indicates relatively uniform pixel values across the image.

The variance is a measure of the dispersion Similar to the standard deviation, a higher variance indicates more diverse pixel values in the images, while

a lower variance suggests more uniform pixel values. Variance can be helpful in understanding the overall distribution of pixel values and identifying regions with significant variations.

3.5.2 EDA Results for GCDB Dataset

1. Basic Data Statistics

The basic data statistics provide crucial insights into the dataset's characteristics. By printing the total number of images, the number of unique labels, and the label mapping, we gain an initial understanding of the dataset's scope and diversity. These statistics help in assessing whether the dataset is adequately representative and balanced, which is crucial for the reliability of subsequent analysis and model performance. Moreover, they assist in confirming that the data loading and preprocessing steps were executed correctly.

Results are as below:
Total images: 10696

Number of unique labels: 24

Label mapping: { 'ALPHA': 1, 'BETA': 2, 'DELTA': 3, 'EPSILON': 4, 'FI': 5, 'GAMMA': 6, 'HETA': 7, 'IOTA': 8, 'KAPA': 9, 'KSI': 10, 'LAMDA': 11, 'MU': 12, 'NU': 13, 'OMEGA': 14, 'OMIKRON': 15, 'PII': 16, 'PSI': 17, 'RO': 18, 'SIGMA': 19, 'TAU': 20, 'THETA': 21, 'XI': 22, 'YPSILON': 23, 'ZETA': 24 }

2. Sample Image Visualization

Visualizing sample images from each label offers an intuitive grasp of the dataset's content. Fig 4: Shows the Sample images from GCDB Datasets. These visualizations allow us to observe the diversity in writing styles, sizes, and other variations within and across characters. By showcasing a few representative images for each label, we can identify any potential data quality issues, outliers, or anomalies that might affect model training. It also enables us to make qualitative judgments about the dataset's suitability for the intended analysis or application.

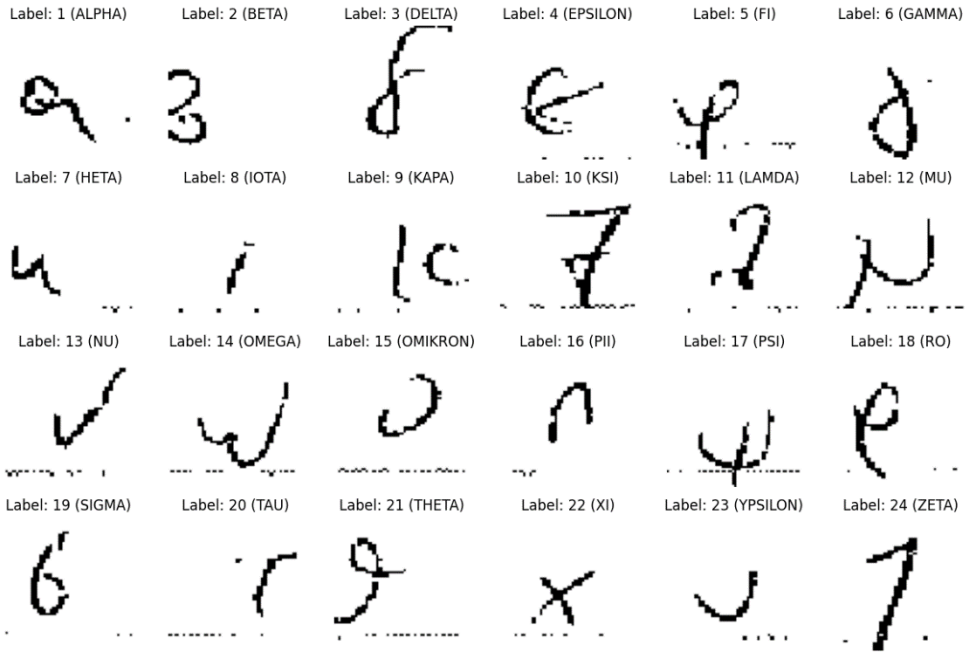


Fig 4. Sample images from GCDB Dataset

3. Label Distribution Plot

The label distribution plot provides a graphical representation of the frequency of each character label in the dataset. This plot is essential for assessing the class balance, which can significantly impact the performance of machine learning models. An imbalanced distribution could lead to biased model predictions.

By visualizing the distribution, we can identify any classes that are overrepresented or underrepresented and decide on appropriate strategies for handling class imbalances during model training.

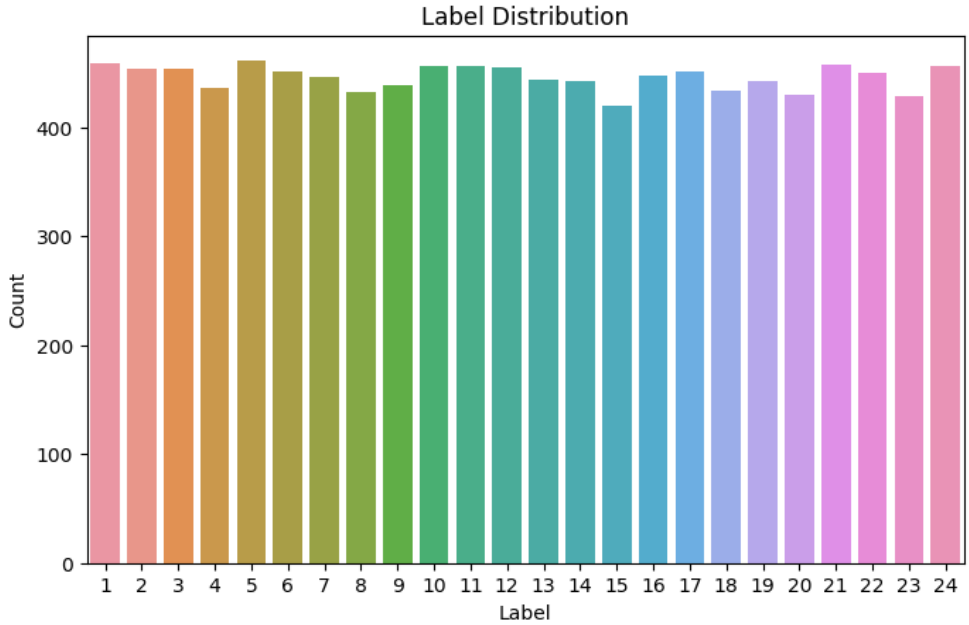


Fig 5: Label Distribution plot For GCDB Dataset

4. Applying PCA

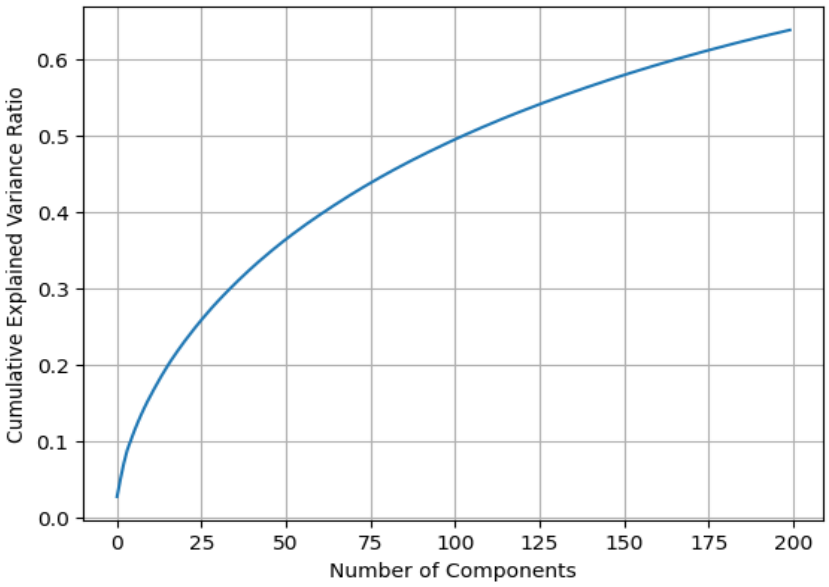


Fig 6: Explained variance ratio

Explained Variance: The explained variance ratio for each principal component indicates the proportion of the total variance captured by that component. A higher explained variance suggests that the corresponding principal component retains more information from the original data. The cumulative explained variance curve illustrates how much variance is retained as the number of principal components increases. As we can see in the Fig 6. more the half of the variance is captured in the data.

Scatter Plots: The scatter plot is a graphical representation that displays the distribution of data points in a two-dimensional plane. In this case, I am visualizing the relationships between the first four principal components (PC_1, PC_2, PC_3, and PC_4) of my dataset. Each point on the scatter plot corresponds to an observation in the reduced-dimensional space defined by these principal components.

If groups of points are tightly clustered together As shown in Fig 7 on the scatter plot, it indicates that these observations are similar in terms of their values along the chosen principal components.

The shape of the histogram provides information about the spread or distribution of values for each principal component. For example, a symmetric bell-

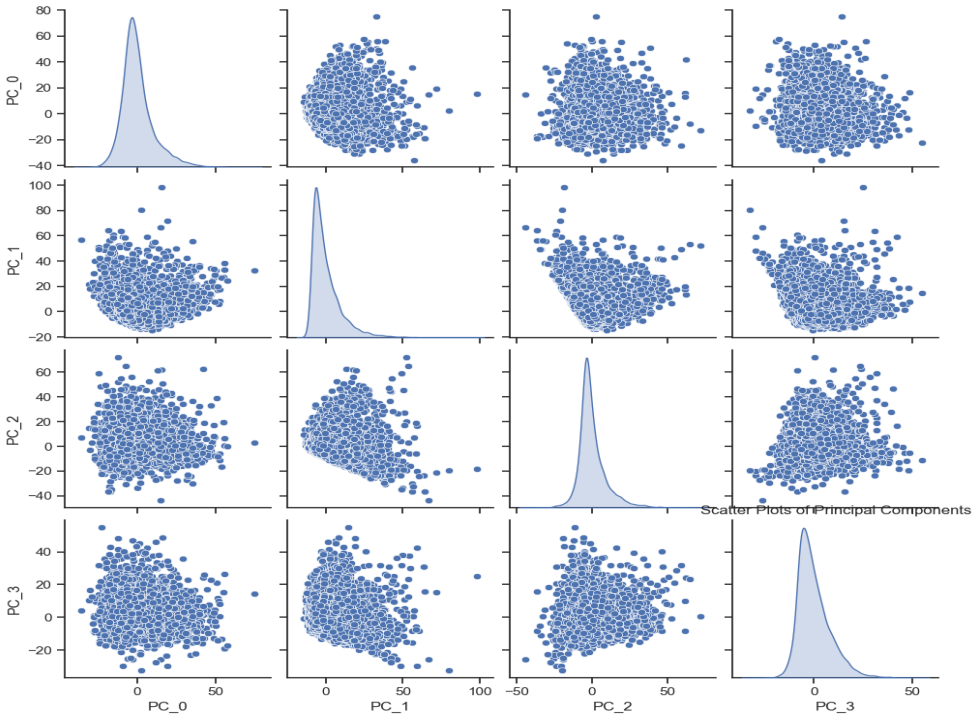


Fig 7: Scatter Plot for First 4 Principal components for GCDB

shaped curve suggests a normal distribution, while skewed shapes indicate asymmetry. The central tendency of the values is reflected in the position of the peak of the histogram. The peak represents the most common or typical value range for the component.

Correlation Matrix: The correlation matrix reveals relationships among the first 10 principal components. Entries along the diagonal (e.g., PC_0 with PC_0) are 1, as they represent self-correlations. Values close to 0 signify weak or no linear correlation, while values further from 0, regardless of sign, denote stronger correlations. This matrix helps identify components that capture similar variance patterns, aiding in conditionality reduction decisions and interpretation of their underlying meaning in the dataset.

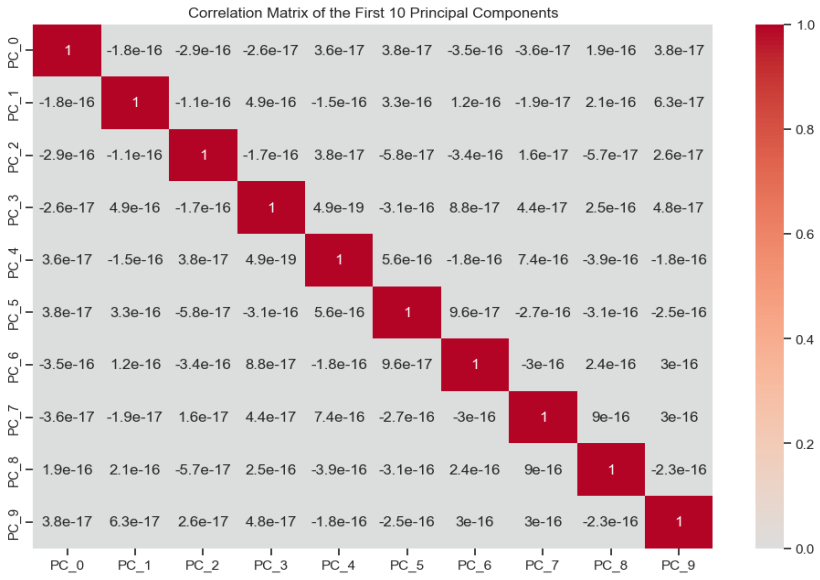


Fig 8: Corelation matrix for GCDB Dataset

4 Experiment and Result

4.1 Applying K-Nearest Neighbour

1. Data Preparation: The first step in applying the KNN algorithm is to prepare the data. This involves splitting the dataset into training, validation, and test sets. The training set is used to train the KNN model, the validation set is used to find the optimal value of K (number of neighbors), and the test set is used to evaluate the final performance of the model.

The first step is to load the training and test data from their respective CSV files into DataFrame objects. This is achieved using a function (load_data) that reads the CSV files and creates the DataFrames train_data and test_data.
Below is the pseudo code for reference:

```
# Load Greek classification dataset
Load train_data from 'path/to/Greek_dataset_1st/train.csv'
Load test_data from 'path/to/Greek_dataset_1st/test.csv'

Extract features X_train_full, X_test from train_data and
test_data
Extract labels y_train_full, y_test from train_data and
test_data

# Load GCDB dataset
Load pca_output_data from 'path/to/pca_output.csv'

Extract features X_pca from pca_output_data
Extract labels y_pca from pca_output_data

# Split data into training, validation, and test sets
X_train, X_val, y_train, y_val =
train_val_test_split(X_train_full, y_train_full,
val_size=0.1, test_size=0.1, random_state=3105)
X_train_pca, X_val_pca, x_test_pca, y_train_pca, y_val_pca,
y_test_pca = train_val_test_split(X_pca, y_pca, val_size=0.1,
test_size=0.1, random_state=3105)
```

2. Choosing the Number of Neighbors (K): The choice of the number of neighbors (K) is critical in KNN. A small K can lead to overfitting, while a large K might lead to underfitting. To determine the optimal K, a common practice is to evaluate the model's performance on the validation set for different values of K. The value of K that yields the highest validation accuracy with a significant difference in train accuracy is chosen.

Below is the pseudo code for reference.

```
Function find_best_K(X_train, y_train, X_val, y_val, max_k,
title):
    best_K = None
    highest_val_accuracy = 0.0
    min_train_accuracy_difference = 0.1

    train_accuracies = Empty List
    val_accuracies = Empty List

    For K in range(1 to max_k):
        y_train_pred = predict(K, X_train, X_train, y_train)
        train_error_rate, train_accuracy =
accuracy_calculate(y_train_pred, y_train)
        y_val_pred = predict(K, X_val, X_train, y_train)
        val_error_rate, val_accuracy =
accuracy_calculate(y_val_pred, y_val)

        If val_accuracy > highest_val_accuracy and
train_accuracy < 1.0 and train_accuracy > 1.0 -
min_train_accuracy_difference:
            best_K = K
            highest_val_accuracy = val_accuracy

        Append train_accuracy to train_accuracies
        Append val_accuracy to val_accuracies

        Print "K =", K, "Train Accuracy =", train_accuracy,
"Validation Accuracy =", val_accuracy

    Plot Line Chart with x-axis as range(1 to max_k), y-axis
as train_accuracies, label="Train Accuracy"
    Add Line Chart with x-axis as range(1 to max_k), y-axis
as val_accuracies, label="Validation Accuracy"
    Set x-axis label to "K"
    Set y-axis label to "Accuracy"
    Set title of the plot to title
    Display legend
    Save plot as an image

    Print "Best K value based on modified criteria:", best_K
    Return best_K
End Function
```

3. Distance Calculation and Neighbor Selection: For a new input data point, the distances between this point and all points in the training set are calculated. Common distance metrics include Euclidean distance. The K nearest neighbors is selected based on the calculated distances.

```
# Function to calculate Euclidean distance between two points
function calculate_distance(point1, point2):
    sum_of_squares = 0
    for i in range(len(point1)):
```

```
        sum_of_squares += (point1[i] - point2[i])^2
    distance = sqrt(sum_of_squares)
    return distance

# Function to predict labels using k-nearest neighbors
function predict(K, x_test, X_train, y_train):
    predictions = []
    for i in range(len(x_test)):
        distances = []
        for j in range(len(X_train)):
            distance = calculate_distance(x_test[i],
X_train[j])
            distances.append((distance, y_train[j]))
        distances.sort()  # Sort distances in ascending order
        neighbors = distances[:K]
        neighbor_labels = [neighbor[1] for neighbor in
neighbors]
        predicted_label = majority_voting(neighbor_labels)
        predictions.append(predicted_label)
    return predictions
```

4. Majority Voting for Classification: For classification tasks, the most common class among the K nearest neighbors is determined. This class is assigned to the new data point as its predicted class.

In the Majority Voting process, each nearest neighbor's label is considered, and a count is maintained for each unique label. The label that appears most frequently among the neighbors is chosen as the predicted label for the new data point. This approach leverages the idea that the label most commonly present within the neighbors is likely to be the correct classification for the new data point.

Below is the pseudo code for reference:

```
# Function to perform majority voting and predict the label
function majority_voting(neighbor_labels):
    label_count = {}
    for label in neighbor_labels:
        if label in label_count:
            label_count[label] += 1
        else:
            label_count[label] = 1
    most_common_label = None
    max_count = 0
    for label, count in label_count.items():
        if count > max_count:
            max_count = count
            most_common_label = label
    return most_common_label
```

4.2 Applying Decision Trees and Random Forest

As same as the KNN model Decision trees are also implemented from scratch while to implement random forest regressor we have used sk-learn model.

Below are the steps which I have considered.

1. Data Preparation: The Data preparation is as same as I have implemented in KNN, For Greek Classification Dataset training data loaded from 'train.csv', and features (X_train) and labels (y_train) were separated. Testing data loaded from 'test.csv', and features (X_test) and labels (y_test) were separated.

For GCDB Dataset raw data was read from 'pca_output.csv'. Later the data is split into training and testing sets using a 80-20 ratio and a fixed random seed. The training set is then used for finding the best parameter using cross validation.

Below is the pseudo code for reference:

```
# Load and split data for the first dataset
Load train_data from 'path/to/train.csv'
Load test_data from 'path/to/test.csv'

Extract features X_train, X_test from train_data and
test_data
Extract labels y_train, y_test from train_data and test_data

# Split data for the second dataset
Load data from 'path/to/pca_output.csv'
Split data into X_train, X_test, y_train, y_test using
train_test_split
```

2. Hyperparameter Tuning and Model Training: In this section, I perform hyperparameter tuning and train the models. I define a range of max_depth values to tune the hyperparameter. Then, I initialize lists to store accuracy scores and CV scores. I iterate through the max_depth range, and within each iteration, I use a Stratified K-Fold Cross-Validation with 5 folds. For each fold, I train a Decision Tree and a Random Forest model. I calculate and store the accuracy scores on both the training and validation sets for each model and fold.

After completing the cross-validation loop, you calculate the average CV scores for both the Decision Tree and Random Forest models. I also determine the best parameters for each model based on the highest CV score obtained. Finally, I train new instances of the Decision Tree and Random Forest models using the best parameters on the full training data.

Below is the pseudo code for reference.

```
# Initialize variables
max_depth_range = range(1, 21)
```

```
kf = StratifiedKFold(n_splits=5, shuffle=True,
random_state=42)

# Initialize lists to store accuracy and CV scores
dt_train_accuracies = [], dt_test_accuracies = []
rf_train_accuracies = [], rf_test_accuracies = []
dt_cv_scores = [], rf_cv_scores = []
best_dt_params = None, best_rf_params = None

# Hyperparameter tuning and cross-validation loop
For each max_depth in max_depth_range:
    Initialize variables to accumulate accuracy scores

    For each train_index, test_index in kf.split(X_train,
y_train):
        Split data into training and validation sets

        Train a Decision Tree model
        Predict on training and validation sets
        Calculate and store accuracy scores

        Train a Random Forest model
        Predict on training and validation sets
        Calculate and store accuracy scores

        Calculate CV scores for this fold

    Calculate average CV scores for Decision Tree and Random
Forest
    Update best_dt_params and best_rf_params if needed

    Train Decision Tree and Random Forest models with best
parameters on full training data
```

3. Model Evaluation and Visualization: I've plotted graphs showing the relationship between the max_depth hyperparameter and the train/test accuracies for both the Decision Tree and Random Forest models. This helps us to understand how increasing the max_depth affects the model's accuracy.

Next, I have generated and displayed combined confusion matrices for both models. These matrices show the number of true positive, true negative, false positive, and false negative predictions, which are essential for understanding the model's predictive performance across different classes.

I calculated and visualized class-wise accuracies for both models, showing how well each class is predicted individually. This information can be useful for identifying which classes the models perform well on and which might need improvement.

Finally, the overall performance metrics such as the best parameters selected, the CV scores obtained during hyperparameter tuning are displayed.

Below is the pseudo code for reference:

```
# Plot train/test accuracies for both models
Plot graphs of max_depth vs. train/test accuracies

# Generate and display combined confusion matrices
Generate confusion matrices for Decision Tree and Random
Forest predictions
Display confusion matrices side by side

# Calculate and visualize class-wise accuracies
Calculate accuracies for each class separately
Plot bar chart of class-wise accuracies for both models

# Display best parameters, CV scores, and other information
Display best_dt_params, best_rf_params
Display CV scores for Decision Tree and Random Forest
Display any additional relevant information
```

4.3 Applying Convolutional Neural Network

1. Data Preparation: `data_dir` is defined as the path to this directory. Two lists, `image_paths` and `labels`, are created to store image file paths and their corresponding labels. These lists will be populated during the data gathering process.

The code iterates through subfolders in the data directory. For each subfolder, it navigates through image files and constructs the full path for each image. These paths and their corresponding labels are appended to the respective lists.

The `train_test_split` function is used twice to create training, validation, and test sets. This helps prevent overfitting by training on one subset and validating/testing on others.

The `ImageDataGenerator` from TensorFlow is used to perform tasks like rescaling pixel values and applying data augmentation techniques (rotation, etc.). These generators facilitate efficient batch-wise loading of data for training and evaluation.

The `ImageDataGenerator`'s `flow_from_dataframe` method is used to load and organize data from dataframes. The dataframes are constructed from the previously created lists of image paths and labels. The generator's parameters, such as target size, batch size, and class mode, are specified to prepare the data for training and evaluation.

Below is the pseudo code for reference.

```
Define the path to the main dataset directory.
Create two empty lists: image_paths and labels.
Loop through subfolders to gather image paths and labels.
    For each subfolder:
        Loop through images in the subfolder:
            Construct image path.
```

```
            Append image path to image_paths.
            Append corresponding label to labels.
Split image_paths and labels into train and test using
train_test_split.
Further split train into train and validation sets.
Convert train_labels, val_labels, and test_labels to string
format.
Create an ImageDataGenerator for data normalization and
augmentation.
Load train, validation, and test data using
flow_from_dataframe method.
Set target size, batch size, and class mode.
```

2. CNN Model Architecture: A sequential model is created using TensorFlow's `Sequential` class. The model consists of convolutional layers, pooling layers, fully connected layers, and activation functions. Below Fig 9: Shows the CNN Model Architecture which I have implemented.

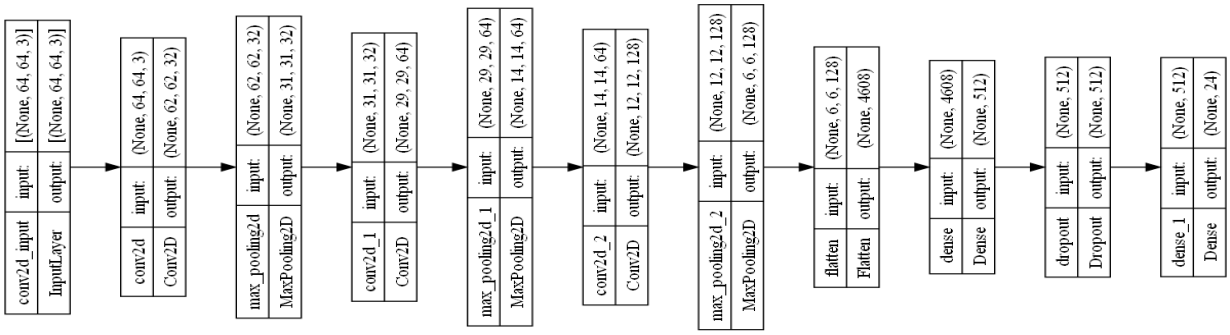


Fig 9: CNN Model Architecture

Below is a pseudo code for reference:

```
Create a Sequential model.
Add Conv2D layer with 32 filters, kernel size (3, 3), and
'relu' activation.
Add MaxPooling2D layer with pool size (2, 2).
Add Conv2D layer with 64 filters and 'relu' activation.
Add MaxPooling2D layer with pool size (2, 2).
Add Conv2D layer with 128 filters and 'relu' activation.
Add MaxPooling2D layer with pool size (2, 2).
Add Flatten layer.
Add Dense layer with 512 units and 'relu' activation.
Add Dropout layer with rate 0.5.
Add Dense layer with num_classes units and 'softmax'
activation.
```

3. Evaluation and Visualization: The plot_training_curves function is defined to visualize the model's training and validation performance. It plots

accuracy and loss curves over epochs to monitor how well the model is learning. This function helps detect overfitting or underfitting.

Model compilation involves specifying the optimizer, loss function, and evaluation metrics. The chosen optimizer (Adam) adjusts the model's weights during training to minimize the loss.

Early stopping is employed to prevent overfitting. It monitors the validation loss during training and halts training if the loss does not improve for a specified number of epochs. This prevents the model from memorizing the training data and improves generalization.

The trained model is evaluated on the test set using the `evaluate` method. The test loss and accuracy are reported, giving an indication of how well the model performs on unseen data.

The `plot_training_curves` function is called to visualize the training and validation accuracy and loss curves. This aids in understanding the learning process and identifying trends or issues.

The confusion matrix is computed using the model's predictions and the true labels of the test set. The seaborn library is used to create a heatmap visualization of the confusion matrix, aiding in the identification of misclassifications.

Below is the pseudo code for reference.

```
Define plot_training_curves function to plot accuracy and
loss curves.
Compile model and plot its architecture.
Rotate the plot image for better visibility.
Set up early stopping callback and train the model.
Evaluate model on the test set and print test loss and
accuracy.
Call plot_training_curves function to visualize training and
validation curves.
Predict class probabilities for the test set.
Convert predicted probabilities to class labels.
Create and plot a confusion matrix for evaluation.
```

4.4 Results

4.4.1 Evaluation Of KNN

1. Greek Classification Dataset

As mentioned before, we have implemented the model from scratch, I have split the data for both the datasets on train, test and validation sets. The model is first train on train and validated on validation set iteratively foy K ranging from 1-20. Below is the

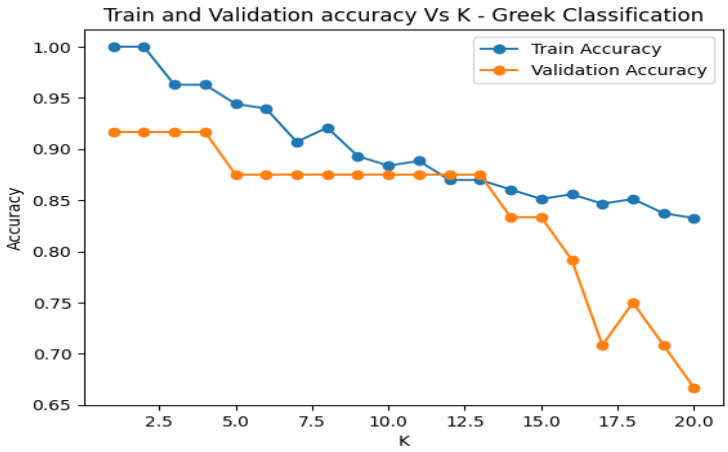


Fig 10: Training and Validation Accuracies for Greek Classification Dataset

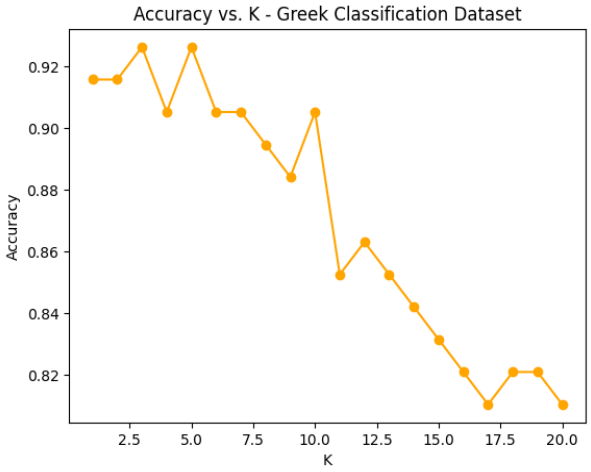
From the above plot we can say that The KNN algorithm achieves perfect train accuracy (1.0000) for both K=1 and K=2. This is because the algorithm assigns each point to its nearest neighbor in the training set, resulting in correct classifications for the training data. However, perfect accuracy on the training data does not necessarily indicate that the model will generalize well to new, unseen data. In other words, for a lower value of K the model may capture noise while training but it may not generalize well for unseen data, In short, the model can overfit.

As K increases from 1, the validation accuracy experiences fluctuations. Initially, at K=1 and K=2, the validation accuracy is high at 0.9167, but as K increases, the validation accuracy remains relatively stable. This suggests that the model is not overfitting to the training data, as indicated by similar train and validation accuracies.

The plot helps us identify the point at which the validation accuracy is maximized while still maintaining a reasonably high train accuracy. This "sweet spot" indicates the value of K that provides a good balance between bias and variance in the model. In this case, K=3 appears to be the optimal choice, as it achieves a validation accuracy of 0.9167 while the train accuracy is still reasonably high at 0.9628.

As K continues to increase beyond the optimal point, the validation accuracy starts to decline. This might be attributed to the fact that higher values of K lead to smoother decision boundaries, causing the model to become overly biased and unable to capture finer patterns in the data. This results in reduced performance on both training and validation data. This indicates that the model starts to underfit the data.

After Finding the best hyperparameter for the model which is K=3 , we have tested the model on test data and Final Test Accuracy with Best K (3) for Greek Classification Dataset: 0.9263



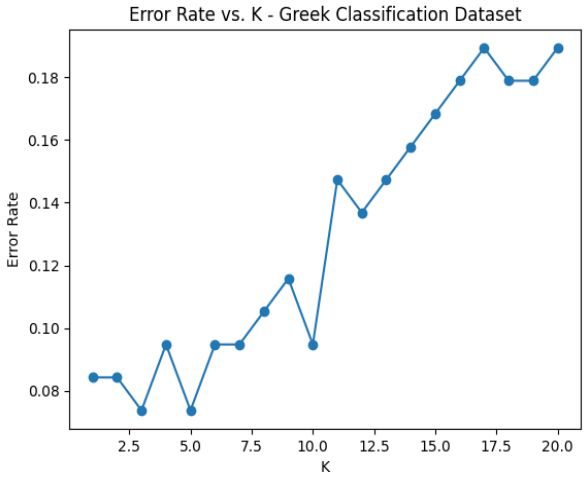


Fig 11 Accuracy Vs K

Fig12: Error Rate Vs K

Above I have plotted the error rate vs K and accuracy vs K for K ranging from 1-20 without the validation set i.e. he model is trained and the model is tested directly on test data.

From the above plots we cat deduce that, at K=1, the error rate is low. This is expected since each instance is classified based on its nearest neighbor, often resulting in accurate classifications. However, as K increases, the error rate tends to rise.

Higher K values lead to a smoother decision boundary, which may lead to higher error rates. This happens because as K increases, the model considers a larger number of neighbors, potentially diluting the ability to capture fine-grained patterns in the data.

The KNN algorithm tends to achieve high accuracy at lower values of K, such as K=1 and K=2. This is because the model "memorizes" the training data and assigns instances to their nearest neighbors.

Balancing the bias and the variance is crucial at this point. As K increases, the model becomes less sensitive to noise in the training data. However, this may also lead to over smoothing and a decrease in accuracy. The plot helps in identifying the range of K values where accuracy remains relatively high and stable.

Below is the confusion matrix build for Greek Classification Dataset:

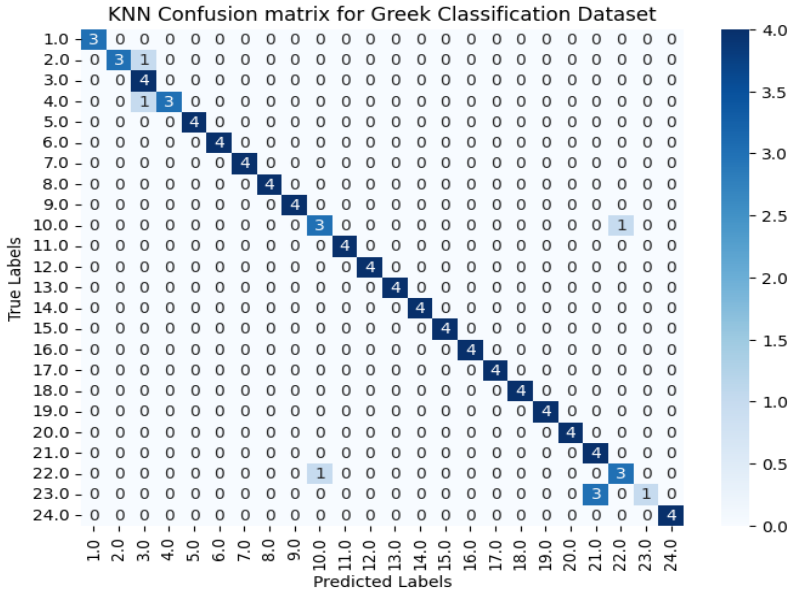


Fig 13: Confusion Matrix for Greek Classification Dataset

The confusion matrix illustrates the model's performance across various classes. The diagonal elements represent the correct classifications, while off-diagonal elements indicate instances of misclassification. Notably, misclassifications are observed between adjacent classes such as (2,3) and (4,3), indicating challenges in distinguishing similar classes. We can see that the huge mistake has been made while classifying the 23rd label.

2. GCDB Dataset

The Same approach has been taken for this data set as above. Below is the plot for training and validation accuracies for GCDB Dataset

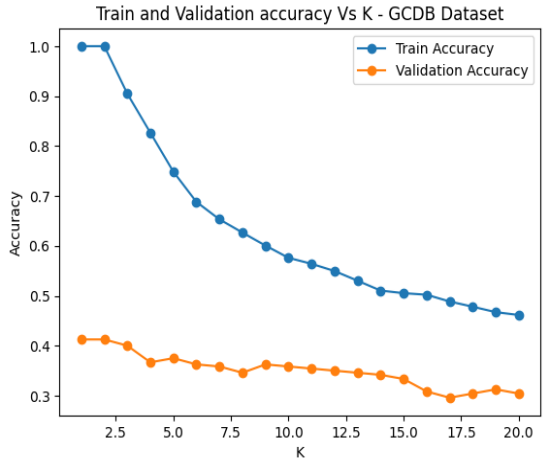


Fig 14: Training and Validation accuracies vs K for GCDB dataset

From the above plot we can say that, with very low values of K (K = 1 and 2), the model achieves perfect training accuracy (1.0000), but the validation accuracy is substantially lower (around 41.25%). This suggests overfitting, where the model memorizes the training data and doesn't generalize well to unseen data.

As K increases, the validation accuracy initially drops. This indicates that when K is very low, the model captures noise and individual data points, leading to poor generalization. However, there is a turning point where validation accuracy starts to increase slightly.

Around K = 3, the validation accuracy is higher than for K = 1 and 2. This suggests that a small neighbourhood of data points helps the model generalize better. It balances the bias (underfitting) introduced by considering too many neighbors and the variance (overfitting) caused by considering too few.

As K keeps increasing, the model becomes simpler. It focuses more on the overall trends in the data and pays less attention to individual data points. However, if K becomes too high, the model may oversimplify and miss important patterns in the data. The model fails to generalize on the GCDB Dataset as for an Optimal K the test accuracy is just 0.3 444.

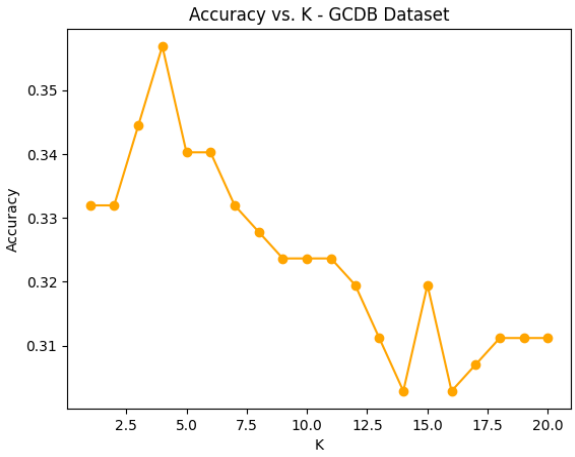


Fig 15 Accuracy Vs K For GCDB

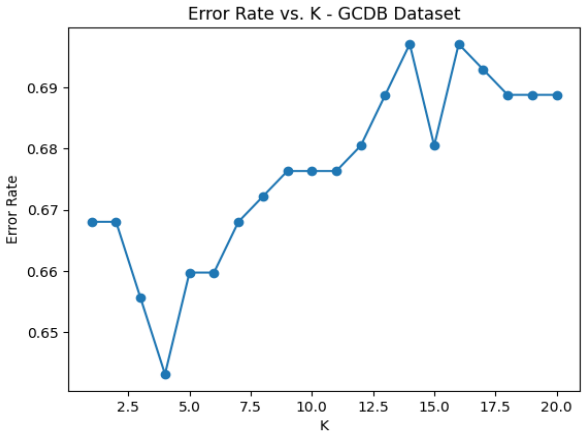


Fig 16: Error Rate Vs K For GCDB

For the error rate plot, at small K values, the model overfits, capturing noise and leading to high error rates on both training and validation data. As K increases, the error rate decreases but model still fails to generalize. Beyond this, the error rate rises as the model underfits, losing discriminative power and encountering difficulties with new data. .

The accuracy vs K curve, Initially, with low K values, the model overfits, resulting in low accuracy. With increasing K, accuracy rises as generalization

improves. However, after reaching an apex, accuracy declines due to underfitting. The model's oversimplification fails to capture data intricacies. This underfitting scenario arises from excessively large K, hindering accurate classification.

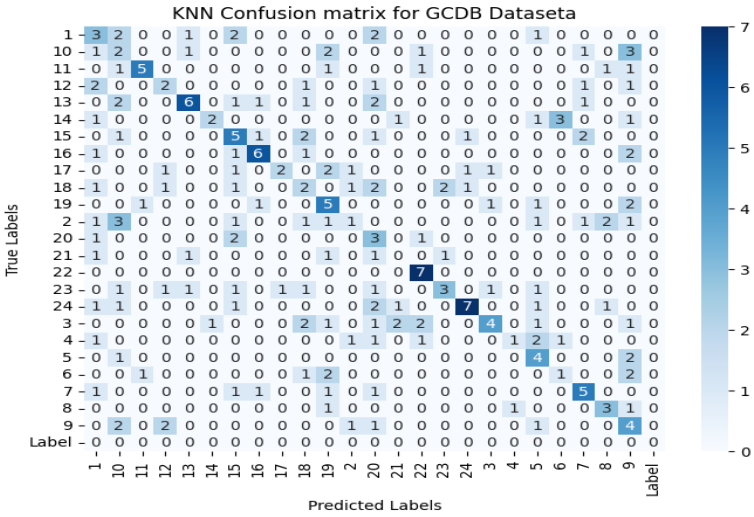


Fig 17: Confusion Matrix for GCDM Dataset

From the above matrix we can say that the model is making mistakes in almost all the classes. It shows a little improvement in predicting the 11,13,19,22, and the 24 class, still it is not enough to call the model generalized for this dataset.

4.4.2 Evaluation of Decision Trees and Random Forest

1. Greek Classification Dataset

As the maximum depth of both Decision Trees and Random Forests increases, the training accuracy generally improves. This is expected, as deeper trees can fit the training data more closely.

However, the test accuracy may not always improve with increasing max depth. The test accuracy initially increases with depth, but after a certain point, it starts to level off or even decline. This indicates the model might be overfitting the training data. This behaviour is shown in the below plot (Fig 18) for both Decision Trees and Random Forest.

Random Forest consistently outperforms Decision Trees across different max depths in terms of both training and test accuracy. This is because Random Forest combines multiple decision trees, reducing the overfitting problem inherent in single Decision Trees.

Fig 18: Training and test Accuracy vs Max Depth for Greek Classification

From the above plot we can see that the gap between training and test accuracies is narrower for Random Forest compared to Decision Trees. This indicates that Random Forest is better at generalizing to new data.

Decision trees creates deep, intricate, and detailed structures that can perfectly fit the training data. This leads to high variance, meaning they can be sensitive to small fluctuations in the training data, including noise and outliers resulting a good accuracy in training data.

The test accuracy indicates how well the models generalize to unseen data. From the plot we can say that the Decision trees are overfitting the data with test accuracy being just 63.33% while Random Forest might be overfitting but it is also generalizing well on the new unseen data with the test accuracy 86.67% for the best max depth which is 13 for decision trees and 14 for random forest.

I have also plotted confusion matrix for both decision trees and random forest which can be seen in Fig 19.

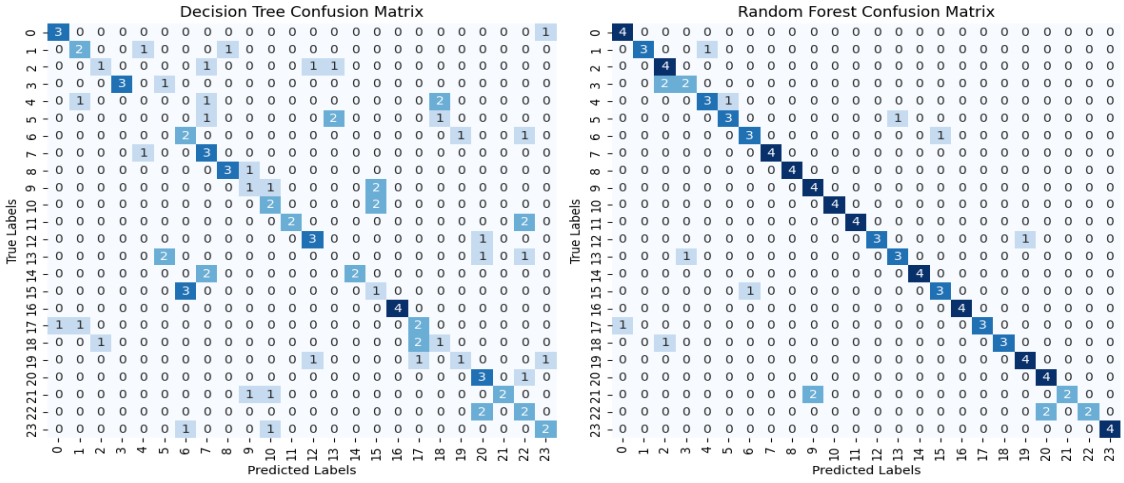


Fig 19: Confusion matrix of Decision trees and Random Forest for Greek Classification Dataset

As seen in the above Decision trees are making mistakes in almost all the classes, as we have discussed earlier, decision trees are very much sensitive to outliers and noises in training data as we build a deeper model but it fails to

generalize on the new unseen data while on the other hand , after combining the power of multiple decision trees i.e random forest we can see that the model is generalizing on the unseen data, there are mistakes as seen from the confusion matrix for predicting the class 9,13,14 and 20,21 and 22.

I have also calculated and plotted the class wise accuracy for both the models which is as shown below:

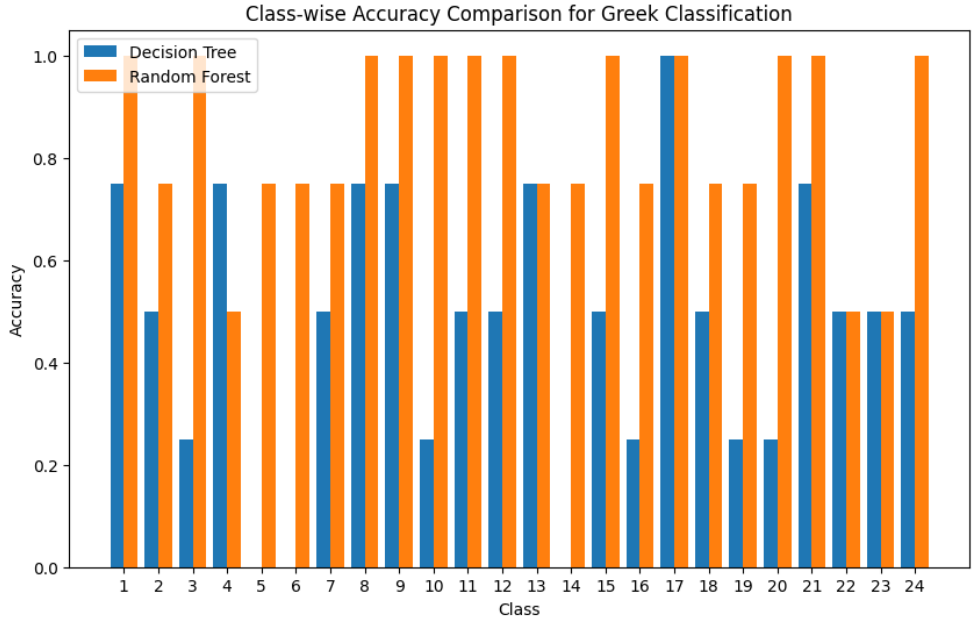


Fig 20: Class wise accuracy for Greek Classification Dataset

We can see that the random forest classifier is consistently outperforming the decision trees for predicting each class.

Finally, as the best max_depth for decision trees is 13 and the best max_depth for random forest is 14 we have evaluated the models based on these parameters and the test accuracy is 0.46875 for decision trees and 0.84375 for random Forest.

2. GCDB Dataset

We have performed the same evaluation matrix for GCDB dataset as we have performed for the Greek Classification Dataset. Below is the plot for Training and cross validation test accuracies vs max depth.

As the max depth increases from 1 to 4, both the train and test accuracies of both the Decision Tree and Random Forest models steadily increase. This indicates that the models are learning from the data and their ability to fit the training data improves. However, there is still a considerable gap between train and test accuracies, suggesting that the models might not be fully utilizing the information and are not yet overfitting.

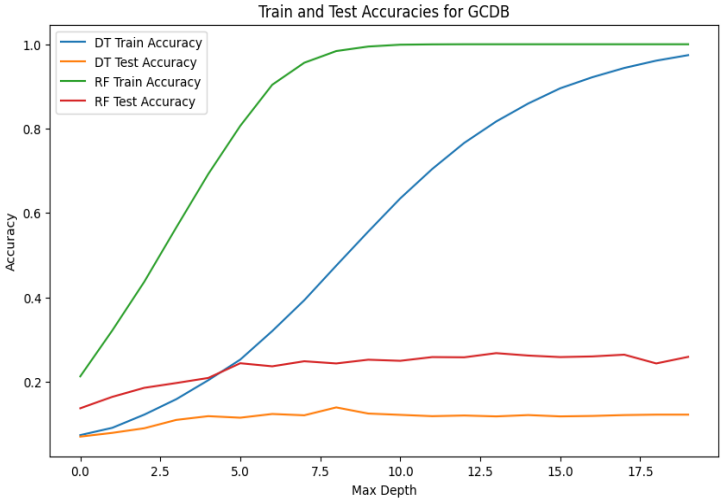


Fig 21: Training and test Accuracy vs Max Depth for GCDB Dataset

Starting from max depth 5, we see a divergence between the train and test accuracies for both models. The train accuracies continue to rise, while the test accuracies plateau or even decrease slightly. This is a classic sign of overfitting. The Decision Tree and Random Forest models are becoming more complex (deeper) and capturing noise and outliers in the training data, which doesn't generalize well to unseen test data. The test accuracies tend to stabilize and might even drop slightly, indicating that the models are becoming too specific to the training data and not generalizing well to new data points.

At max depth 20, we observe an interesting behaviour. The train accuracies are nearly 1.0 for both models, indicating that they are capable of perfectly fitting the training data. However, the test accuracies are not as high as we might expect. This is an example of underfitting. The models are overly complex and might be capturing noise or inconsistencies in the training data, making them perform poorly on the test data. They are essentially memorizing the training data rather than learning meaningful patterns.

In summary we can say that the model is severely overfitting the data. In random forest, as seen in the plot, at max_depth 5 we can see a slight improvement but it again falls down, not generalizing well enough on the new unseen data.

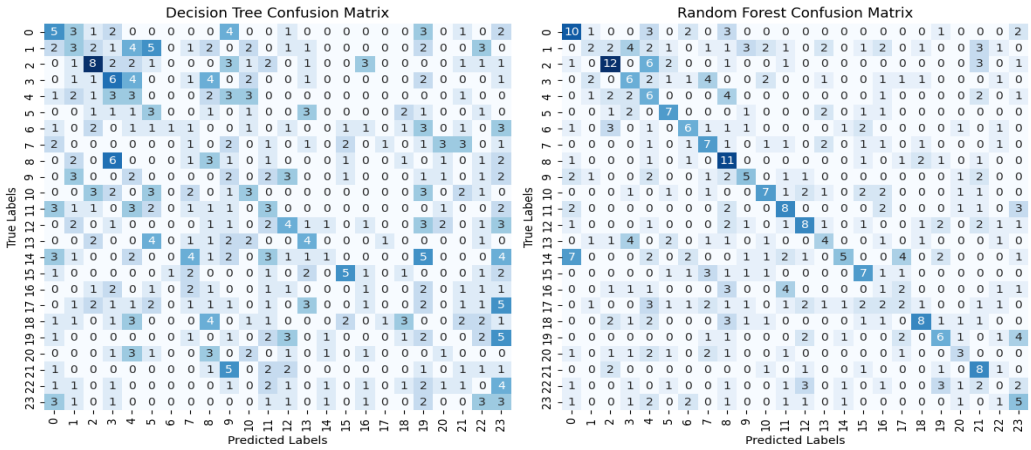


Fig 22: Confusion matrices for GCDB Datasets

Above are the confusion matrices for both the models for GCDB Dataset. As seen from the above plot there are lots of false positives and false negatives in both the matrix.

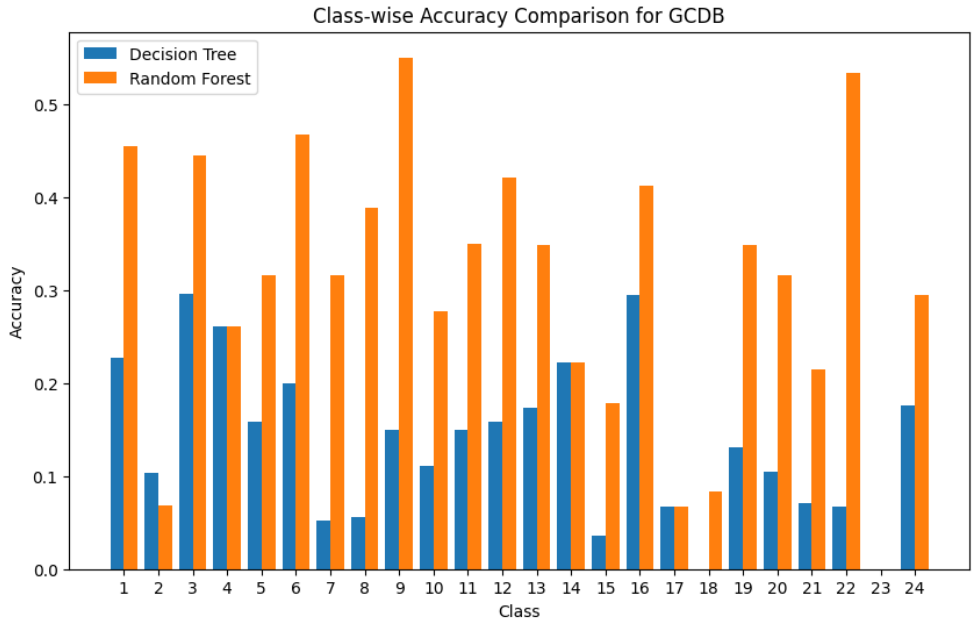


Fig 23: Class wise accuracy for GCDB Dataset

From the above two plots we can see that the model is fail to generalize on GCDB Datasets. We can see in the graphs that the random forest model is performing well than the decision trees but still its not enough, as the numbers are suggesting that random forest also fails to generalize on the new unseen data.

From our experiment, after cross validation we have found out the best parameters for the GCDB dataset is max_depth = 9 for decision trees and max depth = 20 for random forest.

Their corresponding training accuracies are 0.1375 for decision trees and 0.30 for random forest which is an indication of overfitting.

As all the model we have worked with by far i.e. K-Nearest Neighbour, Decision trees and Random Forest, from our experiment we have seen that these models are not generalizing well on the new unseen data for the GCDB data set while they are having a compelling performance on the Greek Classification Dataset.

This is happening because, as mentioned in the start, The data quality for the GCDB dataset is far worse than the Greek classification Dataset. The simple machine learning models might fail to distinguish between certain Greek characters like φ with ψ for example as a result we are not able to get desired results for the Greek Classification dataset.

We are going to experiment with the Convolutional Neural Network, for the GCDB dataset to find out if the CNN can be able to predict the label for such low-quality dataset or not.

4.4.3 Evaluation of Convolutional Neural Network.

The code provided is for building, training, and evaluating a Convolutional Neural Network (CNN) model using the TensorFlow framework. The CNN is designed to classify images from the GCDB dataset, which consists of 24 different classes.

The CNN model architecture is defined using the Sequential API from Keras. It consists of multiple convolutional and pooling layers followed by fully connected layers. The model ends with a SoftMax layer for multiclass classification. The complexity of the model increases with each layer, allowing it to capture hierarchical features.

The model is compiled using the Adam optimizer and categorical cross-entropy loss. An Early Stopping callback is set up to monitor the validation loss and restore the best weights when necessary.

The training and validation accuracy and loss curves are plotted using the `plot_training_curves` function. These curves give insight into the model's performance during training and whether it's overfitting or underfitting.

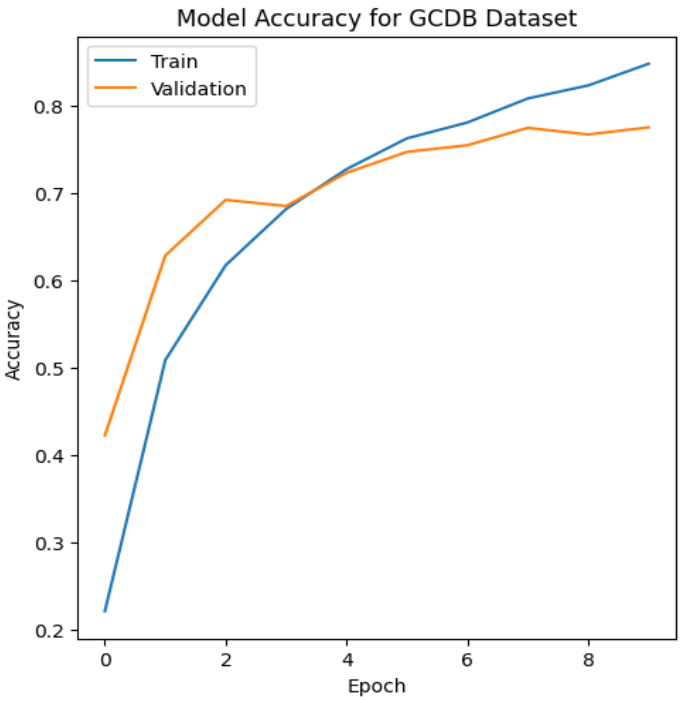


Fig 24: Training and validation Accuracy

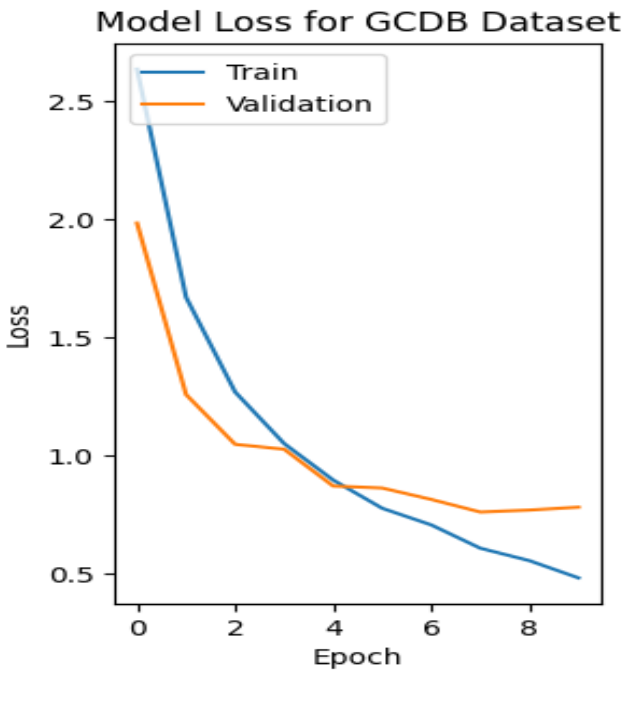


Fig 25: Training and Validation Loss

In the above plot's, the blue line represents the accuracy of the model on the training data. As the epochs increase, the train accuracy improves gradually. This indicates that the model is learning and adapting to the training data.

The orange line represents the accuracy of the model on the validation data. Initially, the validation accuracy increases significantly, suggesting that the model is generalizing well to unseen data. However, after a certain point, the validation accuracy plateaus and starts to show smaller improvements or even slight fluctuations.

Interpretation for Accuracy Plot:

- **Epochs 1-2:** The model is in its early stages of training. Both the train and validation accuracies are relatively low. The model is learning basic patterns from the data.
- **Epochs 3-4:** Both train and validation accuracies are increasing noticeably. The model is capturing more complex patterns and improving its performance on both the training and validation sets.
- **Epochs 5-7:** The validation accuracy continues to improve, but the rate of improvement is slowing down. The model is still learning and adapting to the data, but the improvements are becoming smaller.
- **Epochs 8-10:** The validation accuracy continues to increase slightly, although it may start to plateau. This suggests that the model is approaching its limit in terms of learning from the available data.

Overall, the train and validation accuracy plot show that the model is learning and improving its accuracy on both the training and validation data. However, there are signs that the model's performance might saturate, indicating that further improvements could be limited.

The train and validation loss plot displays the loss values (categorical cross-entropy) achieved by the model on both the training and validation datasets as training progresses over the epochs.

The blue line represents the loss of the model on the training data. As the epochs increase, the train loss decreases. This indicates that the model is getting better at minimizing the difference between predicted and actual values on the training data.

The orange line represents the loss of the model on the validation data. Similar to accuracy, the validation loss initially decreases significantly, showing that the model is generalizing well. However, after a point, the validation loss might stabilize or increase slightly.

Interpretation for Loss Plot:

- **Epochs 1-2:** The model starts with high training and validation losses, which is expected as it's in the early stages of learning**.**
- **Epochs 3-4:** The training and validation losses decrease rapidly, indicating that the model is effectively reducing errors and improving its predictions.

- **Epochs 5-7:** Both training and validation losses continue to decrease, but the rate of improvement slows down. This means the model is making fewer significant corrections.
- **Epochs 8-10:** The training loss continues to decrease, but the validation loss might start to plateau or show slight fluctuations. This suggests that while the model is improving on the training data, it might not generalize as well to new, unseen data.

I have plotted a confusion matrix to get a detailed view of how many instances are correctly classified.

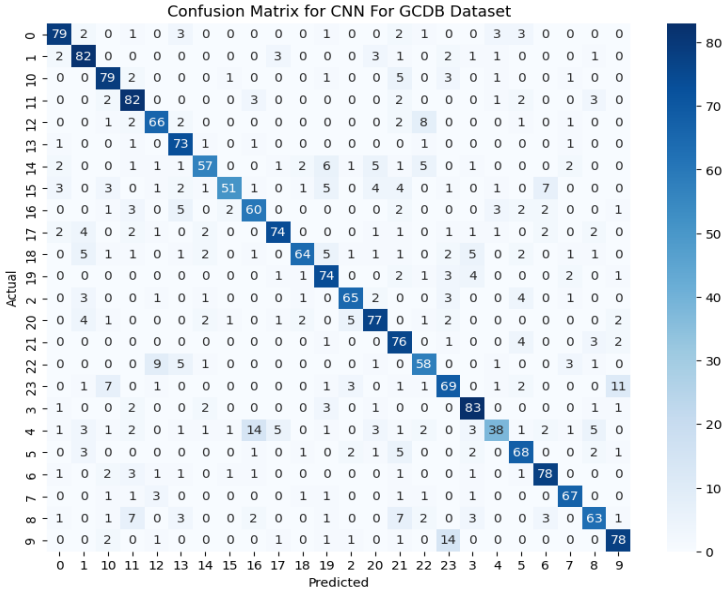


Fig 26: Confusion Matrix of CNN Model For GCDB Dataset

From the above matrix we can say that CNN performs very much well than KNN, Decision trees and Random Forest Classifier classifying almost all of the instances correctly with a test accuracy of 0.7748. Model is making error the most for classifying the 4th class and the 9th class.

Finally, the model is evaluated on test data and the final results are as follows:

	Final Accuracy	**Final Loss**
Train	0.8479	0.4808
Validation	0.7751	0.7802
Test	0.7748	0.8045

Table 1: Final Evaluation of CNN Model

4.4.4 Conclusion

In this dissertation, I have detailed comparisons between two distinct datasets, the Greek Classification Dataset and the GCDB Dataset, that has been carried out to evaluate the performance of machine learning models. The findings from this comparative analysis provide valuable insights into the impact of dataset quality and quantity on model performance.

The Greek Classification Dataset, characterized by its high-quality data and substantial volume of images, demonstrated remarkable suitability for machine learning models. The availability of well-structured and diverse images within this dataset facilitated the successful training and generalization of various machine learning models. The models displayed a strong ability to learn meaningful patterns from the data, resulting in satisfactory performance on both training and validation sets. The dataset's characteristics, such as sufficient data samples per class and balanced distribution, contributed to the model's robustness and capability to generalize effectively to unseen data.

On the contrary, the GCDB Dataset shows a distinct set of challenges due to its large number of images coupled with varying data quality. The models trained on the GCDB Dataset displayed a tendency to overfit, struggling to generalize effectively to new and unseen data. This limitation could be attributed to the dataset's inherent noise, inconsistencies, and imbalances, which affected the models' ability to learn relevant patterns and relationships. Despite the overfitting phenomenon observed in traditional machine learning models, the application of Convolutional Neural Networks (CNN) proved beneficial in mitigating these challenges.

To address the complexities posed by the GCDB Dataset, a CNN model was employed to achieve improved results. The CNN's hierarchical feature extraction, coupled with its ability to learn spatial hierarchies, enabled it to capture intricate patterns within the images. This inherent capacity of CNNs allowed the model to generalize more effectively, yielding better performance on the test dataset compared to traditional machine learning models. The CNN's capability to learn meaningful features without relying heavily on hand-crafted features or explicit data preprocessing rendered it a suitable approach for addressing the inherent challenges of the GCDB Dataset.

Finally, Below Tables shows the summary of the experiment:

Greek Classification Dataset:

	Training Accuracy	**Test Accuracy**	**Model Performance**
KNN	0.9628	0.9263	Generalized
Decision Trees	1.0000	0.46875	Overfitted
Random Forest	1.0000	0.84375	Generalized

GCDB Dataset:

	Training Accuracy	**Test Accuracy**	**Model Performance**
KNN	0.9047	0.3444	**Overfitted**
Decision Trees	0.4751	0.1375	Underfitted
Random Forest	0.9839	0.30	Overfitted
CNN	0.8479	0.7748	Generalized

Table 2 and 3: Model Performances on Greek Classification and GCDB Dataset

4.4.5 Future directions

In addition to the current study, there are several intriguing ideas for further exploration that could enhance the understanding and application of machine learning techniques for Handwritten Greek character classification. One potential extension involves the development of a sophisticated software application that capitalizes on the insights gained from this study. This software could seamlessly integrate image recognition capabilities with machine learning algorithms to accurately classify Greek characters, providing users with valuable insights into the character's identity and the associated accuracy of each classification algorithm.

The envisioned software would leverage the power of machine learning models trained on well-structured datasets, such as the Greek Classification Dataset, to accurately classify Greek characters from user-provided images. This software would be designed to take an input image containing a Greek character and, using the pre-trained models, classify the character with high accuracy. Additionally, the software would present users with a comprehensive report detailing the classification accuracy of each algorithm applied to the given input.

Such a software application could find utility in various domains, including language studies, linguistics, and education. For instance, language enthusiasts and learners could utilize this tool to enhance their understanding of the Greek language by conveniently identifying and verifying Greek characters. Linguistic researchers might employ the software to expedite the process of character recognition in historical documents or inscriptions. Moreover, educators could integrate this tool into language courses to provide students with instant feedback on their character recognition skills, fostering a more engaging learning experience.

To realize this software, a user-friendly interface could be developed, allowing users to effortlessly upload images, view the classification results, and access the accuracy scores of each algorithm employed. The software's success would hinge on the integration of robust image preprocessing, feature extraction, and classification procedures, as well as a well-organized and updatable database of trained models.

Such software has the potential to bridge the gap between advanced machine learning methodologies and practical use cases, ultimately benefiting individuals and communities interested in the accurate classification of Greek characters.

5 Professional Issues and Self-Assessment

1.Professional Issues

Hardware Incompatibility and Data Volume: During the course of the dissertation, significant challenges arose due to hardware incompatibility with the large dataset used for training and evaluation. The absence of a suitable GPU led to prolonged execution times for complex algorithms, impacting both experimentation and result interpretation. This challenge hindered the seamless execution of computationally intensive tasks and necessitated optimization strategies to manage time and resources effectively.

Identifying Relevant Research Papers: One challenge encountered was the identification of relevant research papers that closely aligned with the focus of the study, which was Greek character classification. Locating comprehensive and up-to-date research that addressed the nuances of the Greek language and its character set was critical for gaining insights, validating methodologies, and contextualizing the study within the broader research landscape.

2.Self-Assessment

The project had two goals, to deepen my knowledge of machine learning and to make a contribution, to the field of character recognition technology. It demanded not expertise, but also critical thinking, problem solving abilities and a strong dedication, to excellence. As I immersed myself in the intricacies of data preparation selecting the algorithms and evaluating models, I consistently faced challenges that pushed me to think creatively and make informed choices. Completing this project has not provided me with knowledge, about machine learning and character recognition but has also set the groundwork for future advancements in this field. In the following section I will discuss the projects structure, goals and importance offering an understanding of its impact, on computer science and machine learning.

6 References

[1] Sai Jahnavi Bachu, "Character Recognition using KNN Algorithm," International Journal of Science and Research (IJSR), Volume 10, Issue 4, April 2021.link

[2] Deepu V., Sriganesh M., Ramakrishnan A. G. "Principal Component Analysis for Online Handwritten Character Recognition." Conference Paper, August 2004. DOI: 10.1109/ICPR.2004.1334196 Source: IEEE Xplore. link

[3] Md. Anwar Hossain & Md. Mohon Ali, "Recognition of Handwritten Digit using Convolutional Neural Network (CNN),".link

[4] Paraskevi Platanou, John Pavlopoulos, and Georgios Papaioannou, "Handwritten Paleographic Greek Text Recognition: A Century-Based Approach," Proceedings of the 13th Conference on Language Resources and Evaluation (LREC 2022), pages 6585–6589, Marseille, 20-25 June 2022.link

[5] D. V. Deepu, M. Sriganesh, and A. G. Ramakrishnan, "Principal Component Analysis for Online Handwritten Character Recognition," in *Proceedings of the IEEE International Conference on Pattern Recognition (ICPR)*, August 2004, DOI: 10.1109/ICPR.2004.1334196.link

[6] Classification of Handwritten Greek Letters - https://www.kaggle.com/datasets/katianakontolati/classification-of-handwritten-greek-letters

[7] Handwritten Greek Characters from GCDB - https://www.kaggle.com/datasets/vrushalipatel/handwritten-greek-characters-from-gcdb/code

[8] Wikipedia page for KNN- https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#cite_note-1

[9] Wikipedia page for PCA -[1] https://en.wikipedia.org/wiki/Principal_component_analysis

[10] Wikipedia page for Manhattan Distance- https://en.wikipedia.org/wiki/Taxicab_geometry
[11] Wikipedia page for decision trees – https://en.wikipedia.org/wiki/Decision_tree_learning
[12] Wikipedia page for Ensembled Learning- https://en.wikipedia.org/wiki/Ensemble_learning
[13]Scikit Learn Module for ensembled learning- https://scikit-learn.org/stable/modules/ensemble.html
[14] Wikipedia Page for CNN- https://en.wikipedia.org/wiki/Convolutional_neural_network
[15] Wikipedia Page for PCA - https://en.wikipedia.org/wiki/Principal_component_analysis

7 How to Run This Project.

I have provided a .zip file, this file contains a folder named My_project.

This folder will consist of all the scrips to reproduce all the results and plot which I have already explained in this dissertation.

Initially the folder structure should be looking like below.

```
My_project/
│
├─── Greek_dataset/
├─── GCDB_dataset/
├─── Preprocessing.py
├─── EDA_Greek_Classification.py
├─── EDA_GCDB_Dataset.py
├─── PCA.py
├─── KNN.py
├─── Decision_Trees.py
├─── Tensorflow_GCDB_Dataset.py
├─── master.py
│
├─── accuracy_vs_k_GCDB_Dataset.png
├─── accuracy_curves_CNN.png
├─── KNN_confusion_matrix_GCDB_Dataset.png
├─── ...
│
├─── dissertation_report/
│    ├─── Handwritten_Greek_Character_Recognition_Dissertation.pdf
```

System Requirements:

Operating System: Windows 10 or Higher

Python version: 3.10 or higher

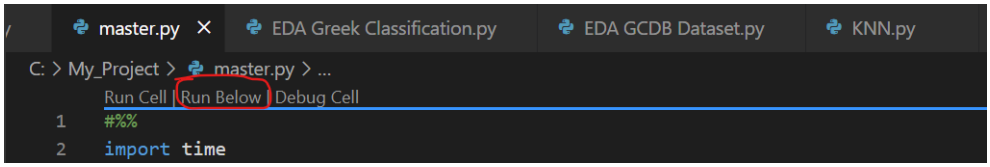
Python Libraries: numpy, pandas, sk-learn, matplotlib, seaborn, time, os, warnings, random, PIL, git, shutil, TensorFlow

Visual Studio Code

In order to execute this project it is essential to extract the .zip file in **"C:/"** drive.

There are mainly 3 ways to run this project:

1. Open master.py in visual studio code and click on "Rub below" as shown in the below fig.



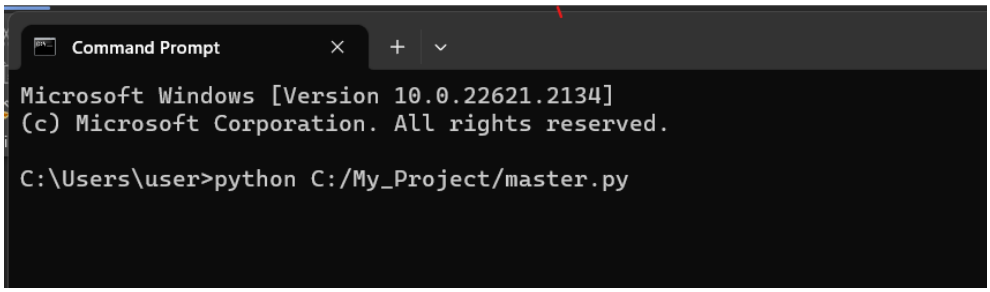
As soon as this is done, this script will execute all the .py files which is required to generate the plots. The plots will be saved at the location "C:\My_Project".

2. One can open the script on visual studio code and run it the same way as mentioned above or run it in visual studio terminal. It will reproduce the same results in the above-mentioned location.
One can open the script in any other compiler and run it in a different specified why. It will still run without error and generate the results.

3. The third way is to open a command prompt (cmd) in machine and simply run the below command. One can also run individual script to reproduce the results.

command: python C:/My_Project/master.py

This action required python to be installed already on the machine.
Below is a demo of the command.



8 Appendix

8.1 Scripts

8.1.1 master.py Python Script

```
#%%
import time

# List of script file paths
script_files = [
    "C:/My_Project/Preprocessing.py",
    "C:/My_Project/EDA Greek Classification.py",
    "C:/My_Project/EDA GCDB Dataset.py",
    "C:/My_Project/PCA.py",
    "C:/My_Project/KNN.py",
    "C:/My_Project/descision trees.py",
    "C:/My_Project/Tensorflow GCDB Dataset.py"
]

# Execute each script using exec()
total_execution_time = 0

for script_file in script_files:
    print(f"Executing {script_file}...")
    start_time = time.time()
    with open(script_file, "r") as file:
        script_content = file.read()
        exec(script_content)
    end_time = time.time()

    execution_time = end_time - start_time
    total_execution_time += execution_time
    print(f"{script_file} executed in {execution_time:.2f}
seconds")

total_execution_time_minutes = total_execution_time / 60
print("All scripts executed successfully")
print(f"Total execution time:
{total_execution_time_minutes:.2f} minutes")
```

8.1.2 preprocessing.py Python script

```
#%%
import os
import pandas as pd
from PIL import Image
from git import Repo
import shutil
import random
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Define the mapping of characters to labels
label_mapping = {
    'ALPHA': 1, 'BETA': 2, 'GAMMA': 3, 'DELTA': 4, 'EPSILON':
5, 'ZETA': 6, 'HETA': 7, 'THETA': 8, 'IOTA': 9, 'KAPA': 10,
    'LAMDA': 11, 'MU': 12, 'NU': 13, 'KSI': 14, 'OMIKRON': 15,
'PII': 16, 'RO': 17, 'SIGMA': 18, 'TAU': 19, 'YPSILON': 20,
    'FI': 21, 'XI': 22, 'PSI': 23, 'OMEGA': 24
}

# Remove the existing directory if it exists
local_repo_path = "C:/My_Project/Greek_dataset/"
if os.path.exists(local_repo_path):
    # Remove files within .git directory manually
    git_dir = os.path.join(local_repo_path, ".git")
    for root, dirs, files in os.walk(git_dir, topdown=False):
        for file in files:
            file_path = os.path.join(root, file)
            os.chmod(file_path, 0o777)  # Change file
permission to be writable
            os.remove(file_path)

    # Remove the entire directory
    shutil.rmtree(local_repo_path)

# Clone the GitHub repository to the local directory
repo_url = "https://github.com/Samay3131/Greek_Dataset.git"  #
Replace with your GitHub repository URL
Repo.clone_from(repo_url, local_repo_path)

git_dir = os.path.join(local_repo_path, ".git")
```

```
for root, dirs, files in os.walk(git_dir, topdown=False):
    for file in files:
        file_path = os.path.join(root, file)
        os.chmod(file_path, 0o777)  # Change file permission to
be writable
        os.remove(file_path)
os.chmod(git_dir, 0o777)
shutil.rmtree(git_dir)
# Initialize an empty list to store the data

data = []

# Loop through each folder and process the images
folder_path = local_repo_path

for folder_name in os.listdir(folder_path):
    label = label_mapping.get(folder_name, 0)  # Set the label
to 0 if folder_name not found in the mapping
    folder_full_path = os.path.join(folder_path, folder_name)
    image_files = [file for file in
os.listdir(folder_full_path) if file.lower().endswith(('.png',
'.jpg', '.bmp', '.gif'))]
    selected_images = random.sample(image_files,
min(len(image_files), 100))

    for image_file in selected_images:
        image_path = os.path.join(folder_full_path, image_file)


        # Check if the current item is a file and skip if it's
not an image file
        if not os.path.isfile(image_path) or not
image_file.lower().endswith(('.png', '.jpg', '.bmp', '.gif')):
            continue

        image = Image.open(image_path).convert("L")  # Convert
image to grayscale

        # Resize the image to 14x12 (196 pixels) and flatten it
to get a 1D array
        resized_image = image.resize((14, 12))
        pixel_values = list(resized_image.getdata())
```

```
        # Add the label to the end of the pixel_values list
        pixel_values.append(label)

        data.append(pixel_values)

# Convert the data list to a pandas DataFrame
df = pd.DataFrame(data)

# Save the DataFrame to a CSV file
csv_file_path = "C:/My_Project/output.csv"  # Replace with the
desired output CSV file path
if os.path.exists(csv_file_path):
    os.remove(csv_file_path)
    df.to_csv(csv_file_path, index=False, header=False,
float_format='%.2f', mode='w')
else:
     df.to_csv(csv_file_path, index=False, header=False,
float_format='%.2f', mode='w')
##################################################################
###################################

local_repo_path_2 = "C:/My_Project/Greek_dataset_1st/"

if os.path.exists(local_repo_path_2):
    # Remove files within .git directory manually
    git_dir = os.path.join(local_repo_path_2, ".git")
    for root, dirs, files in os.walk(git_dir, topdown=False):
        for file in files:
            file_path = os.path.join(root, file)
            os.chmod(file_path, 0o777)  # Change file
permission to be writable
            os.remove(file_path)
    shutil.rmtree(local_repo_path_2)

repo_url_2 =
"https://github.com/Samay3131/Greek_dataSet_1st.git"  # Replace
with your GitHub repository URL
Repo.clone_from(repo_url_2, local_repo_path_2)

git_dir = os.path.join(local_repo_path_2, ".git")
for root, dirs, files in os.walk(git_dir, topdown=False):
    for file in files:
        file_path = os.path.join(root, file)
```

```
        os.chmod(file_path, 0o777)  # Change file permission to
be writable
        os.remove(file_path)
os.chmod(git_dir, 0o777)
shutil.rmtree(git_dir)
```

8.1.3 EDA Greek Classification.py Python script

```
#%%

import os
import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

train_folder =
'C:/My_Project/Greek_dataset_1st/train_letters_images/'
test_folder =
'C:/My_Project/Greek_dataset_1st/test_letters_images/'

# Load a few images from the train and test folders
train_images = os.listdir(train_folder)
test_images = os.listdir(test_folder)

print(f"Number of train images: {len(train_images)}")
print(f"Number of test images: {len(test_images)}")


# Common size for resizing
target_size = (100, 100)  # Adjust the size as needed

# Initialize list to store image data
images = []

# Loop through the image files in the train folder
for filename in os.listdir(train_folder):
    if filename.endswith('.jpg'):  # Assuming images are in
.jpg format
        image_path = os.path.join(train_folder, filename)
```

```
        # Load and resize the image
        img = Image.open(image_path).resize(target_size)
        img_array = np.array(img)

        images.append(img_array)


# Convert images to a numpy array
images_array = np.array(images)

# Flatten the images
flattened_images = images_array.reshape(images_array.shape[0],
-1)

# Perform k-means clustering
num_clusters = 24
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
clusters = kmeans.fit_predict(flattened_images)

# Visualize images from the first 5 clusters
sample_size = 5
for cluster in range(min(5, num_clusters)):
    cluster_indices = [i for i, c in enumerate(clusters) if c
== cluster]

    if len(cluster_indices) >= sample_size:
        sample_indices = np.random.choice(cluster_indices,
size=sample_size, replace=False)
    else:
        sample_indices = cluster_indices

    plt.figure(figsize=(10, 5))
    for i, index in enumerate(sample_indices):
        img = images_array[index]
        plt.subplot(1, sample_size, i + 1)
        plt.imshow(img, cmap='gray')
        plt.title(f"Cluster {cluster + 1}")
        plt.axis('off')
    plt.show()

from sklearn.manifold import TSNE
import seaborn as sns
```

```
# Perform t-SNE for visualization
tsne = TSNE(n_components=2, random_state=42)
tsne_result = tsne.fit_transform(flattened_images)

# Create a scatter plot to visualize clusters
plt.figure(figsize=(10, 8))
sns.scatterplot(x=tsne_result[:, 0], y=tsne_result[:, 1],
hue=clusters, palette='tab20', legend='full')
plt.title('t-SNE Visualization of Clusters')
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.legend(loc='upper right')
plt.show()


from scipy.stats import norm

# Pixel Distribution Analysis
mean_pixel_values = np.mean(flattened_images, axis=1)
std_pixel_values = np.std(flattened_images, axis=1)
variance_pixel_values = np.var(flattened_images, axis=1)
median_pixel_values = np.median(flattened_images, axis=1)
print("Medain Pixel Values")
print(mean_pixel_values)
print("Standard deviation of pixel Values")
print(std_pixel_values)
print("Varience of pixel values")
print(variance_pixel_values)
print("Medain of Pixel Values")
print(median_pixel_values)
# Create line plots to visualize pixel distribution statistics
with normal-like distribution curves
plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)
plt.hist(mean_pixel_values, bins=20, density=True,
color='blue', alpha=0.7)
x_range = np.linspace(np.min(mean_pixel_values),
np.max(mean_pixel_values), 100)
plt.plot(x_range, norm.pdf(x_range, np.mean(mean_pixel_values),
np.std(mean_pixel_values)), color='red', linewidth=2)
plt.title('Mean Pixel Values')
```

```
plt.subplot(2, 2, 2)
plt.hist(std_pixel_values, bins=20, density=True,
color='green', alpha=0.7)
x_range = np.linspace(np.min(std_pixel_values),
np.max(std_pixel_values), 100)
plt.plot(x_range, norm.pdf(x_range, np.mean(std_pixel_values),
np.std(std_pixel_values)), color='red', linewidth=2)
plt.title('Standard Deviation of Pixel Values')

plt.subplot(2, 2, 3)
plt.hist(variance_pixel_values, bins=20, density=True,
color='purple', alpha=0.7)
x_range = np.linspace(np.min(variance_pixel_values),
np.max(variance_pixel_values), 100)
plt.plot(x_range, norm.pdf(x_range,
np.mean(variance_pixel_values), np.std(variance_pixel_values)),
color='red', linewidth=2)
plt.title('Variance of Pixel Values')

plt.subplot(2, 2, 4)
plt.hist(median_pixel_values, bins=20, density=True,
color='orange', alpha=0.7)
x_range = np.linspace(np.min(median_pixel_values),
np.max(median_pixel_values), 100)
plt.plot(x_range, norm.pdf(x_range,
np.mean(median_pixel_values), np.std(median_pixel_values)),
color='red', linewidth=2)
plt.title('Median Pixel Values')

plt.tight_layout()
plt.show()

# %%
```

8.1.4 EDA GCDB DATAset.py Python Script

```
#%%

import os
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

data = []
labels = []
label_mapping = {}  # To map folder names to labels
data_folder = 'C:/My_Project/Greek_dataset'

# Iterate through the subfolders, where each subfolder
represents a character
for label, char_folder in
enumerate(sorted(os.listdir(data_folder))):
    label += 1  # Start labels from 1
    char_folder_path = os.path.join(data_folder, char_folder)
    if os.path.isdir(char_folder_path):
        label_mapping[char_folder] = label  # Map folder name
to label
        for image_file in os.listdir(char_folder_path):
            image_path = os.path.join(char_folder_path,
image_file)
            image = Image.open(image_path)
            image = image.resize((64, 64))  # Resize image to
64x64
            data.append(np.array(image).flatten())  # Flatten
the image
            labels.append(label)

# Convert lists to numpy arrays
data = np.array(data)
labels = np.array(labels)

# Create a DataFrame for EDA
df = pd.DataFrame(data=data, columns=[f"pixel_{i}" for i in
range(data.shape[1])])
df['Label'] = labels

# Display basic statistics
print("Total images:", len(df))
print("Number of unique labels:", df['Label'].nunique())
```

```
print("Label mapping:", label_mapping)

# Display a few sample images
plt.figure(figsize=(12, 8))
for i in range(len(label_mapping)):
    label = i + 1
    sample_row = df[df['Label'] == label].iloc[0, :-
1].values  # Get pixel values for the sample
    sample_image = sample_row.reshape(64,
64).astype(np.float32) / 255.0  # Adjust pixel values
    plt.subplot(4, 6, label)
    plt.imshow(sample_image, cmap='gray')
    plt.title(f"Label: {label}
({list(label_mapping.keys())[i]})")
    plt.axis('off')
plt.tight_layout()
plt.show()

# Plot label distribution
plt.figure(figsize=(8, 5))
sns.countplot(x='Label', data=df)
plt.title("Label Distribution")
plt.xlabel("Label")
plt.ylabel("Count")
plt.show()

num_components = 200  # Number of principal components
pca = PCA(n_components=num_components)
scaler = StandardScaler()
data_scaled = scaler.fit_transform(df.drop('Label', axis=1))  #
Standardize data
df_pca = pd.DataFrame(pca.fit_transform(data_scaled),
columns=[f'PC_{i}' for i in range(num_components)])
df_pca['Label'] = df['Label']

# Calculate basic statistics for each principal component
pca_stats = df_pca.describe()

# Print the statistics
print(pca_stats)

explained_variance_ratio = pca.explained_variance_ratio_
```

```
cumulative_explained_variance =
np.cumsum(explained_variance_ratio)
plt.plot(cumulative_explained_variance)
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.grid(True)
plt.show()

import seaborn as sns
import matplotlib.pyplot as plt

# Scatter plots of pairs of principal components
sns.set(style="ticks")
sns.pairplot(df_pca.iloc[:, :4], diag_kind="kde")  # Plotting
the first 4 principal components
plt.title("Scatter Plots of Principal Components")
plt.show()

# Calculate correlations between the first few principal
components
num_pc_to_correlate = 10
pc_correlations = df_pca.iloc[:, :num_pc_to_correlate].corr()

# Print the correlations in a table format
print("Correlation Matrix for the First", num_pc_to_correlate,
"Principal Components:")
print(pc_correlations)

import seaborn as sns
import matplotlib.pyplot as plt

# Calculate correlations between the first 10 principal
components
pc_correlations = df_pca.iloc[:, :10].corr()

# Visualize the correlation matrix as a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(pc_correlations, annot=True, cmap='coolwarm',
center=0)
plt.title("Correlation Matrix of the First 10 Principal
Components")
plt.show()
```

8.1.5 PCA.py Python Script

```
#%%
import pandas as pd
from sklearn.decomposition import PCA

# Load the dataset from the CSV file
csv_file_path = "C:/My_Project/output.csv"  # Replace with the
path to your output CSV file
df = pd.read_csv(csv_file_path, header=None)

# Separate features (X) from labels (y)
X = df.iloc[:, :-1]  # All columns except the last one
y = df.iloc[:, -1]   # Last column (labels)

# Perform PCA to determine the number of components to keep
pca = PCA()
pca.fit(X)

# Calculate the cumulative explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = explained_variance_ratio.cumsum()

# Determine the number of components that capture a significant
amount of variance
threshold_variance = 0.95  # You can adjust this threshold
based on your preference
num_components =
len(cumulative_variance_ratio[cumulative_variance_ratio <=
threshold_variance])

print(f"Number of components to retain for {threshold_variance
* 100:.2f}% explained variance: {num_components}")

# Initialize PCA with the determined number of components
pca = PCA(n_components=num_components)

# Perform PCA on the feature matrix X
X_pca = pca.fit_transform(X)

# Convert the PCA results back to a DataFrame
df_pca = pd.DataFrame(X_pca)
```

```
# Add the label column back to the DataFrame
df_pca['Label'] = y

# Save the PCA-transformed data to a new CSV file
pca_csv_file_path = "C:/My_Project/pca_output.csv"  # Replace
with the desired PCA output CSV file path
df_pca.to_csv(pca_csv_file_path, index=False)
```

8.1.6 KNN.py Python Script

```
#%%
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

def train_val_test_split(X, Y, val_size, test_size,
random_state):
    X_train, X_temp, Y_train, Y_temp = train_test_split(X, Y,
test_size=val_size + test_size, random_state=random_state)
    X_val, X_test, Y_val, Y_test = train_test_split(X_temp,
Y_temp, test_size=test_size / (val_size + test_size),
random_state=random_state)
    return X_train, X_val, X_test, Y_train, Y_val, Y_test
# Function to sort an array
def sort_distance(e):
    for i in range(len(e)):
        for j in range(i + 1, len(e)):
            swap = 0
            if e[i] > e[j]:
                swap = e[i]
                e[i] = e[j]
                e[j] = swap
    return e

# Function to find the most common nearest neighbor label
def split_neighbours(pred):
    y_pred = []
    for i in range(len(pred)):
        temp = []
```

```
        temp = list(pred[i])
        c = []
        for i in temp:
            c.append(temp.count(i))
            index_c = c.index(max(c))
        y_pred.append(temp[index_c])
    return y_pred

# Function to calculate accuracy and error rate
def accuracy_calculate(y_pred, y_test):
    acc = y_pred == y_test
    ct = 0
    cf = 0
    for i in acc:
        if i == True:
            ct = ct + 1
        if i == False:
            cf = cf + 1

    accuracy = 1 - (cf / (cf + ct))
    error_rate = 1 - accuracy
    return error_rate, accuracy

# Function to predict labels using k-nearest neighbors
import math as m

def predict(K, x_test, X_train, Y_train):
    pred = []
    for i in range(len(x_test)):
        index = []
        neighbour = []
        distance = []
        dist = []
        for j in range(len(X_train)):
            distance.append(m.sqrt(sum((x_test[i] -
X_train[j])**2)))
        dist.extend(distance)
        eucl = sort_distance(dist)
        for i in range(len(distance)):
            for j in range(len(eucl)):
                if eucl[i] == distance[j]:
                    index.append(j)
        neighbour.append(index[:K])
```

```
        for i in neighbour:
            pred.append(Y_train[i])
    pred = np.asarray(pred)
    y_pred = split_neighbours(pred)
    y_pred = np.asarray(y_pred)
    return y_pred

def find_best_K(X_train, y_train, X_val, y_val, max_k,title):
    best_K = None
    highest_val_accuracy = 0.0
    min_train_accuracy_difference = 0.1  # Minimum difference
between train and validation accuracy

    train_accuracies = []  # List to store train accuracies for
all K values
    val_accuracies = []    # List to store validation
accuracies for all K values

    for K in range(1, max_k + 1):
        # Calculate training and validation accuracy for the
current K
        y_train_pred = predict(K, X_train, X_train, y_train)
        train_error_rate, train_accuracy =
accuracy_calculate(y_train_pred, y_train)
        y_val_pred = predict(K, X_val, X_train, y_train)
        val_error_rate, val_accuracy =
accuracy_calculate(y_val_pred, y_val)

        # Check if the validation accuracy is higher and the
train accuracy is not 100%
        if val_accuracy > highest_val_accuracy and
train_accuracy < 1.0 and train_accuracy > 1.0 -
min_train_accuracy_difference:
            best_K = K
            highest_val_accuracy = val_accuracy

        # Append accuracies to the lists
        train_accuracies.append(train_accuracy)
        val_accuracies.append(val_accuracy)

        # Print the current K and its corresponding train and
validation accuracies
```

```
        print(f"K = {K}: Train Accuracy = {train_accuracy:.4f},
Validation Accuracy = {val_accuracy:.4f}")

    # Plot train and validation accuracies
    plt.plot(range(1, max_k + 1), train_accuracies, marker='o',
label='Train Accuracy')
    plt.plot(range(1, max_k + 1), val_accuracies, marker='o',
label='Validation Accuracy')
    plt.xlabel('K')
    plt.ylabel('Accuracy')
    plt.title(title)
    plt.legend()
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

    # Print the best K value
    print(f"\nBest K value based on modified criteria:
{best_K}")

    return best_K

def plot_error_rate_vs_k(X_train, y_train, X_test, y_test,
max_k, title):
    k_values = range(1, max_k + 1)
    error_rates = []
    for k in k_values:
        y_pred = predict(k, X_test, X_train, y_train)
        error_rate, _ = accuracy_calculate(y_pred, y_test)
        error_rates.append(error_rate)
    plt.plot(k_values, error_rates, marker='o')
    plt.xlabel('K')
    plt.ylabel('Error Rate')
    plt.title(title)
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

def plot_accuracy_vs_k(X_train, y_train, X_test, y_test, max_k,
title):
    k_values = range(1, max_k + 1)
    accuracies = []
    for k in k_values:
        y_pred = predict(k, X_test, X_train, y_train)
        _, accuracy = accuracy_calculate(y_pred, y_test)
```

```
        accuracies.append(accuracy)
    plt.plot(k_values, accuracies, marker='o', color='orange')
    plt.xlabel('K')
    plt.ylabel('Accuracy')
    plt.title(title)
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

def plot_confusion_matrix(y_true, y_pred, labels, title):
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title(title)
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

# Load the Greek classification dataset
train_data =
pd.read_csv('C:/My_Project/Greek_dataset_1st/train.csv')
test_data =
pd.read_csv('C:/My_Project/Greek_dataset_1st/test.csv')

X_train_full = np.asarray(train_data.iloc[:, :-1])
y_train_full = np.asarray(train_data.iloc[:, -1])
X_test = np.asarray(test_data.iloc[:, :-1])
y_test = np.asarray(test_data.iloc[:, -1])

# Set the maximum value for K to 20
max_k = 20
from sklearn.model_selection import train_test_split
# Find the best K value for Greek Classification Dataset
X_train, X_val, y_train, y_val = train_test_split(X_train_full,
y_train_full, test_size=0.1, random_state=3105)
best_K_greek = find_best_K(X_train, y_train, X_val, y_val,
max_k,"Train and Validation accuracy Vs K - Greek
Classification")

# Use the best K value for Greek Classification Dataset to
calculate and print the final test accuracy
```

```
y_pred_best_greek = predict(best_K_greek, X_test, X_train_full,
y_train_full)
final_error_rate_greek, final_accuracy_greek =
accuracy_calculate(y_pred_best_greek, y_test)
print(f"\nFinal Test Accuracy with Best K ({best_K_greek}) for
Greek Classification Dataset: {final_accuracy_greek:.4f}")

# Plot error rate and accuracy vs. K for the Greek
Classification Dataset
plot_error_rate_vs_k(X_train_full, y_train_full, X_test,
y_test, max_k, "Error Rate vs. K - Greek Classification
Dataset")
# Plot accuracy vs. K for the Greek Classification Dataset
plot_accuracy_vs_k(X_train_full, y_train_full, X_test, y_test,
max_k, "Accuracy vs. K - Greek Classification Dataset")

# Plot confusion matrix for Greek Classification Dataset
plot_confusion_matrix(y_test, y_pred_best_greek,
np.unique(y_train_full), "KNN Confusion matrix for Greek
Classification Dataset")

##################################################################
#############
# # Load PCA output data
pca_output_data = pd.read_csv('C:/My_Project/pca_output.csv',
header=None)
X_pca = np.asarray(pca_output_data.iloc[:, :-1])
y_pca = np.asarray(pca_output_data.iloc[:, -1])

# # Split the PCA data into training, validation, and test sets
X_train_pca, X_val_pca, x_test_pca, y_train_pca, y_val_pca,
y_test_pca = train_val_test_split(X_pca, y_pca, val_size=0.1,
test_size=0.1, random_state=3105)

# Find the best K value for PCA output data
best_K_pca = find_best_K(X_train_pca, y_train_pca, X_val_pca,
y_val_pca, max_k,"Train and Validation accuracy Vs K - GCDB
Dataset")

# Use the best K value for PCA output data to calculate and
print the final test accuracy
y_pred_best_pca = predict(best_K_pca, x_test_pca, X_train_pca,
y_train_pca)
```

```
final_error_rate_pca, final_accuracy_pca =
accuracy_calculate(y_pred_best_pca, y_test_pca)
print(f"\nFinal Test Accuracy with Best K ({best_K_pca}) for
PCA Output Data: {final_accuracy_pca:.4f}")

plot_error_rate_vs_k(X_train_pca, y_train_pca, x_test_pca,
y_test_pca, max_k, "Error Rate vs. K - GCDB Dataset")
plot_accuracy_vs_k(X_train_pca, y_train_pca, x_test_pca,
y_test_pca, max_k, "Accuracy vs. K - GCDB Dataset")

# # Plot confusion matrix for PCA output data
plot_confusion_matrix(y_test_pca, y_pred_best_pca,
np.unique(y_pca), "KNN Confusion matrix for GCDB Dataseta")


# # Finally, the code execution ends here
```

8.1.7 decision tress.py Python Script

```
# %%
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import
train_test_split,StratifiedKFold
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
from sklearn.exceptions import UndefinedMetricWarning

# Ignore UndefinedMetricWarning
warnings.filterwarnings("ignore",
category=UndefinedMetricWarning)



class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = {}
```

```
    def gini_index(self, y):
        classes = np.unique(y)
        n = len(y)
        gini = 1.0

        for c in classes:
            p = np.count_nonzero(y == c) / n
            gini -= p ** 2

        return gini

    def best_split(self, X, y):
        n_features = X.shape[1]
        best_gini = float('inf')
        best_feature_index = None
        best_threshold = None

        for feature_index in range(n_features):
            thresholds = np.unique(X[:, feature_index])

            for threshold in thresholds:
                left_indices = X[:, feature_index] <= threshold
                right_indices = X[:, feature_index] > threshold
                gini = (self.gini_index(y[left_indices]) *
np.sum(left_indices) +
                        self.gini_index(y[right_indices]) *
np.sum(right_indices)) / len(y)

                if gini < best_gini:
                    best_gini = gini
                    best_feature_index = feature_index
                    best_threshold = threshold

        return best_feature_index, best_threshold

    def build_tree(self, X, y, depth=0):
        if self.max_depth is not None and depth ==
self.max_depth or len(np.unique(y)) == 1:
            return {'label': np.argmax(np.bincount(y))}

        feature_index, threshold = self.best_split(X, y)
        left_indices = X[:, feature_index] <= threshold
        right_indices = X[:, feature_index] > threshold
```

```
        left = self.build_tree(X[left_indices],
y[left_indices], depth + 1)
        right = self.build_tree(X[right_indices],
y[right_indices], depth + 1)

        return {'feature_index': feature_index, 'threshold':
threshold, 'left': left, 'right': right}

    def fit(self, X, y):
        self.tree = self.build_tree(X, y)

    def predict_sample(self, x, tree):
        while 'feature_index' in tree:


            if x[tree['feature_index']] <= tree['threshold']:
                tree = tree['left']
            else:
                tree = tree['right']
        return tree['label']

    def predict(self, X):
        predictions = []
        for x in X:
            predictions.append(self.predict_sample(x,
self.tree))
        return predictions

    def accuracy(self, y_pred, y_test):
        acc = y_pred == y_test
        ct = 0
        cf = 0
        for i in acc:
            if i == True:
                ct += 1  # calculating the count of True
predicted values
            if i == False:
                cf += 1  # calculating the count of False
predicted values

        accuracy = ct / (cf + ct)
        error_rate = 1 - accuracy
```

```
        return accuracy, error_rate

def data_split(data):
    X = np.asarray(data.iloc[:, :-1]).astype(float)
    y = np.asarray(data.iloc[:, -1]).astype(int)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2,random_state= 3105)

    return X_train, X_test, y_train, y_test

def plot_confusion_matrix(y_true, y_pred, title, filename,
dataset_name):
    num_labels = 24
    cm = confusion_matrix(y_true, y_pred, labels=np.arange(1,
num_labels + 1))
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
cbar=False)
    plt.title(title)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')


    tick_positions = np.arange(0.5, num_labels + 0.5)
    plt.xticks(tick_positions, np.arange(1, num_labels + 1))
    plt.yticks(tick_positions, np.arange(1, num_labels + 1))

    save_path = f"C:/My_Project/{filename}_{dataset_name}.png"
    plt.savefig(save_path)
    plt.show(block=False)




def class_wise_accuracy(y_true, y_pred, y_pred_rf):
    classes = np.unique(y_true)
    accuracies_dt = []
    accuracies_rf = []

    for cls in classes:
        mask = y_true == cls
        y_true_class = y_true[mask]
        y_pred_class = np.array(y_pred)[mask]
```

```
        y_pred_rf_class = y_pred_rf[mask]

        accuracy_dt = accuracy_score(y_true_class,
y_pred_class)
        accuracies_dt.append(accuracy_dt)

        accuracy_rf = accuracy_score(y_true_class,
y_pred_rf_class)
        accuracies_rf.append(accuracy_rf)

    return classes, accuracies_dt, accuracies_rf




def plot_class_wise_accuracy(classes, accuracies_dt,
accuracies_rf, dataset_name):
    plt.figure(figsize=(10, 6))
    plt.bar(classes - 0.2, accuracies_dt, width=0.4,
label='Decision Tree', align='center')
    plt.bar(classes + 0.2, accuracies_rf, width=0.4,
label='Random Forest', align='center')
    plt.xticks(classes)
    plt.xlabel('Class')
    plt.ylabel('Accuracy')
    plt.title(f'Class-wise Accuracy Comparison for
{dataset_name}')
    plt.legend()
    plt.savefig(f'C:/My_Project/class_wise_accuracy_{dataset_na
me}.png')
    plt.show(block=False)

def plot_combined_confusion_matrix(y_true, y_pred_dt,
y_pred_rf, dataset_name):
    num_labels = 24
    cm_dt = confusion_matrix(y_true, y_pred_dt,
labels=np.arange(1, num_labels + 1))
    cm_rf = confusion_matrix(y_true, y_pred_rf,
labels=np.arange(1, num_labels + 1))

    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
```

```
    sns.heatmap(cm_dt, annot=True, fmt='d', cmap='Blues',
cbar=False)
    plt.title("Decision Tree Confusion Matrix")
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')

    plt.subplot(1, 2, 2)
    sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues',
cbar=False)
    plt.title("Random Forest Confusion Matrix")
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')

    plt.tight_layout()
    save_path =
f"C:/My_Project/combined_confusion_matrix_{dataset_name}.png"
    plt.savefig(save_path)
    plt.show()



def model_with_cross_validation(X_train, y_train, X_test,
y_test, max_depth_range, dataset_name):
    kf = StratifiedKFold(n_splits=5, shuffle=True,
random_state=42)
    dt_train_accuracies = []
    dt_test_accuracies = []
    rf_train_accuracies = []
    rf_test_accuracies = []

    dt_cv_scores = []  # New list to store cross-validation
scores for Decision Tree
    rf_cv_scores = []  # New list to store cross-validation
scores for Random Forest
    best_dt_params = None  # Initialize to None
    best_rf_params = None  # Initialize to None

    for max_depth in max_depth_range:
        dt_train_accuracy_sum = 0
        dt_test_accuracy_sum = 0
        rf_train_accuracy_sum = 0
        rf_test_accuracy_sum = 0
```

```
        dt_cv_scores_for_depth = []  # List to store CV scores
for Decision Tree for this depth
        rf_cv_scores_for_depth = []  # List to store CV scores
for Random Forest for this depth

        for train_index, test_index in kf.split(X_train,
y_train):
            X_train_fold, X_test_fold = X_train[train_index],
X_train[test_index]
            y_train_fold, y_test_fold =
y_train[train_index].astype(int),
y_train[test_index].astype(int)

            # Decision Tree
            dt_tree = DecisionTree(max_depth=max_depth)
            dt_tree.fit(X_train_fold, y_train_fold)
            dt_train_y_pred = dt_tree.predict(X_train_fold)
            dt_test_y_pred = dt_tree.predict(X_test_fold)
            dt_train_accuracy, _ =
dt_tree.accuracy(dt_train_y_pred, y_train_fold)
            dt_test_accuracy, _ =
dt_tree.accuracy(dt_test_y_pred, y_test_fold)
            dt_train_accuracy_sum += dt_train_accuracy
            dt_test_accuracy_sum += dt_test_accuracy

            # Random Forest
            rf_classifier =
RandomForestClassifier(n_estimators=100, max_depth=max_depth,
random_state=42)
            rf_classifier.fit(X_train_fold, y_train_fold)
            rf_train_y_pred =
rf_classifier.predict(X_train_fold)
            rf_test_y_pred = rf_classifier.predict(X_test_fold)
            rf_train_accuracy, _ =
dt_tree.accuracy(rf_train_y_pred, y_train_fold)
            rf_test_accuracy, _ =
dt_tree.accuracy(rf_test_y_pred, y_test_fold)
            rf_train_accuracy_sum += rf_train_accuracy
            rf_test_accuracy_sum += rf_test_accuracy

            # Append CV scores for this fold
            dt_cv_scores_for_depth.append(dt_test_accuracy)
```

```
            rf_cv_scores_for_depth.append(rf_test_accuracy)

        dt_train_accuracies.append(dt_train_accuracy_sum /
kf.n_splits)
        dt_test_accuracies.append(dt_test_accuracy_sum /
kf.n_splits)
        rf_train_accuracies.append(rf_train_accuracy_sum /
kf.n_splits)
        rf_test_accuracies.append(rf_test_accuracy_sum /
kf.n_splits)

        # Store average CV scores for this depth
        dt_cv_scores.append(np.mean(dt_cv_scores_for_depth))
        rf_cv_scores.append(np.mean(rf_cv_scores_for_depth))

        print(f"Max Depth: {max_depth}")
        print(f"Decision Tree - Train Accuracy:
{dt_train_accuracy_sum / kf.n_splits:.4f}, Test Accuracy:
{dt_test_accuracy_sum / kf.n_splits:.4f}")
        print(f"Random Forest - Train Accuracy:
{rf_train_accuracy_sum / kf.n_splits:.4f}, Test Accuracy:
{rf_test_accuracy_sum / kf.n_splits:.4f}")
        print("="*50)

        # Store best parameters based on test accuracy
        if best_dt_params is None or
np.mean(dt_cv_scores_for_depth) >
np.mean(dt_cv_scores[best_dt_params['max_depth']]):
            best_dt_params = {'max_depth': max_depth}
        if best_rf_params is None or
np.mean(rf_cv_scores_for_depth) >
np.mean(rf_cv_scores[best_rf_params['max_depth']]):
            best_rf_params = {'max_depth': max_depth}

    print("Decision Tree Cross-Validation Scores:",
dt_cv_scores)
    print("Random Forest Cross-Validation Scores:",
rf_cv_scores)
    print("Best Decision Tree Parameters:", best_dt_params)
    print("Best Random Forest Parameters:", best_rf_params)

    # Predict using best parameters on testing data
```

```
    best_dt_tree =
DecisionTree(max_depth=best_dt_params['max_depth'])
    best_dt_tree.fit(X_train, y_train)
    best_dt_test_y_pred = best_dt_tree.predict(X_test)

    best_rf_classifier =
RandomForestClassifier(n_estimators=100,
max_depth=best_rf_params['max_depth'], random_state=42)
    best_rf_classifier.fit(X_train, y_train)
    best_rf_test_y_pred = best_rf_classifier.predict(X_test)

    dt_test_accuracy, _ =
best_dt_tree.accuracy(best_dt_test_y_pred, y_test)
    rf_test_accuracy, _ =
best_dt_tree.accuracy(best_rf_test_y_pred, y_test)

    print("Decision Tree - Test Accuracy:", dt_test_accuracy)
    print("Random Forest - Test Accuracy:", rf_test_accuracy)

    # Plot train and test accuracies
    plt.figure(figsize=(10, 6))
    plt.plot(dt_train_accuracies, label='DT Train Accuracy')
    plt.plot(dt_test_accuracies, label='DT Test Accuracy')
    plt.plot(rf_train_accuracies, label='RF Train Accuracy')
    plt.plot(rf_test_accuracies, label='RF Test Accuracy')
    plt.xlabel('Max Depth')
    plt.ylabel('Accuracy')
    plt.title(f'Train and Test Accuracies for {dataset_name}')
    plt.legend()
    plt.savefig(f'C:/My_Project/train_test_accuracies_{dataset_
name}.png')
    plt.show()

    plot_combined_confusion_matrix(y_test, best_dt_test_y_pred,
best_rf_test_y_pred, dataset_name)
    classes, accuracies_dt, accuracies_rf =
class_wise_accuracy(y_test, best_dt_test_y_pred,
best_rf_test_y_pred)
    plot_class_wise_accuracy(classes, accuracies_dt,
accuracies_rf, dataset_name)


# Call the function with the testing data
```

```
max_depth_range = range(1, 21)
# Processing First dataset
# Separate features (X) and labels (y)
train_data =
pd.read_csv('C:/My_Project/Greek_dataset_1st/train.csv',
header=None)
test_data =
pd.read_csv('C:/My_Project/Greek_dataset_1st/test.csv',
header=None)
X_train = np.asarray(train_data.iloc[:, :-1]).astype(float)
y_train = np.asarray(train_data.iloc[:, -1]).astype(int)

X_test = np.asarray(test_data.iloc[:, :-1]).astype(float)
y_test = np.asarray(test_data.iloc[:, -1]).astype(int)


model_with_cross_validation(X_train, y_train, X_test, y_test,
max_depth_range, dataset_name="Greek Classification")
################################################################
############################
# Processing Second dataset
data = pd.read_csv('C:/My_Project/pca_output.csv')
X_train, X_test, y_train, y_test = data_split(data)
model_with_cross_validation(X_train, y_train,X_test, y_test,
max_depth_range, dataset_name="GCDB")

# %%
```

8.1.8 Tensorflow GCDB Dataset.py python Script.

```
#%%
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

def train_val_test_split(X, Y, val_size, test_size,
random_state):
    X_train, X_temp, Y_train, Y_temp = train_test_split(X, Y,
test_size=val_size + test_size, random_state=random_state)
```

```
    X_val, X_test, Y_val, Y_test = train_test_split(X_temp,
Y_temp, test_size=test_size / (val_size + test_size),
random_state=random_state)
    return X_train, X_val, X_test, Y_train, Y_val, Y_test
# Function to sort an array
def sort_distance(e):
    for i in range(len(e)):
        for j in range(i + 1, len(e)):
            swap = 0
            if e[i] > e[j]:
                swap = e[i]
                e[i] = e[j]
                e[j] = swap
    return e

# Function to find the most common nearest neighbor label
def split_neighbours(pred):
    y_pred = []
    for i in range(len(pred)):
        temp = []
        temp = list(pred[i])
        c = []
        for i in temp:
            c.append(temp.count(i))
            index_c = c.index(max(c))
        y_pred.append(temp[index_c])
    return y_pred

# Function to calculate accuracy and error rate
def accuracy_calculate(y_pred, y_test):
    acc = y_pred == y_test
    ct = 0
    cf = 0
    for i in acc:
        if i == True:
            ct = ct + 1
        if i == False:
            cf = cf + 1

    accuracy = 1 - (cf / (cf + ct))
    error_rate = 1 - accuracy
    return error_rate, accuracy
```

```
# Function to predict labels using k-nearest neighbors
import math as m

def predict(K, x_test, X_train, Y_train):
    pred = []
    for i in range(len(x_test)):
        index = []
        neighbour = []
        distance = []
        dist = []
        for j in range(len(X_train)):
            distance.append(m.sqrt(sum((x_test[i] -
X_train[j])**2)))
        dist.extend(distance)
        eucl = sort_distance(dist)
        for i in range(len(distance)):
            for j in range(len(eucl)):
                if eucl[i] == distance[j]:
                    index.append(j)
        neighbour.append(index[:K])
        for i in neighbour:
            pred.append(Y_train[i])
    pred = np.asarray(pred)
    y_pred = split_neighbours(pred)
    y_pred = np.asarray(y_pred)
    return y_pred

def find_best_K(X_train, y_train, X_val, y_val, max_k,title):
    best_K = None
    highest_val_accuracy = 0.0
    min_train_accuracy_difference = 0.1  # Minimum difference
between train and validation accuracy

    train_accuracies = []  # List to store train accuracies for
all K values
    val_accuracies = []    # List to store validation
accuracies for all K values

    for K in range(1, max_k + 1):
        # Calculate training and validation accuracy for the
current K
        y_train_pred = predict(K, X_train, X_train, y_train)
```

```
        train_error_rate, train_accuracy =
accuracy_calculate(y_train_pred, y_train)
        y_val_pred = predict(K, X_val, X_train, y_train)
        val_error_rate, val_accuracy =
accuracy_calculate(y_val_pred, y_val)

        # Check if the validation accuracy is higher and the
train accuracy is not 100%
        if val_accuracy > highest_val_accuracy and
train_accuracy < 1.0 and train_accuracy > 1.0 -
min_train_accuracy_difference:
            best_K = K
            highest_val_accuracy = val_accuracy

        # Append accuracies to the lists
        train_accuracies.append(train_accuracy)
        val_accuracies.append(val_accuracy)

        # Print the current K and its corresponding train and
validation accuracies
        print(f"K = {K}: Train Accuracy = {train_accuracy:.4f},
Validation Accuracy = {val_accuracy:.4f}")

    # Plot train and validation accuracies
    plt.plot(range(1, max_k + 1), train_accuracies, marker='o',
label='Train Accuracy')
    plt.plot(range(1, max_k + 1), val_accuracies, marker='o',
label='Validation Accuracy')
    plt.xlabel('K')
    plt.ylabel('Accuracy')
    plt.title(title)
    plt.legend()
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

    # Print the best K value
    print(f"\nBest K value based on modified criteria:
{best_K}")

    return best_K

def plot_error_rate_vs_k(X_train, y_train, X_test, y_test,
max_k, title):
```

```
    k_values = range(1, max_k + 1)
    error_rates = []
    for k in k_values:
        y_pred = predict(k, X_test, X_train, y_train)
        error_rate, _ = accuracy_calculate(y_pred, y_test)
        error_rates.append(error_rate)
    plt.plot(k_values, error_rates, marker='o')
    plt.xlabel('K')
    plt.ylabel('Error Rate')
    plt.title(title)
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

def plot_accuracy_vs_k(X_train, y_train, X_test, y_test, max_k,
title):
    k_values = range(1, max_k + 1)
    accuracies = []
    for k in k_values:
        y_pred = predict(k, X_test, X_train, y_train)
        _, accuracy = accuracy_calculate(y_pred, y_test)
        accuracies.append(accuracy)
    plt.plot(k_values, accuracies, marker='o', color='orange')
    plt.xlabel('K')
    plt.ylabel('Accuracy')
    plt.title(title)
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

def plot_confusion_matrix(y_true, y_pred, labels, title):
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title(title)
    plt.savefig(f'C:/My_Project/{title}.png')
    plt.show(block=False)

# Load the Greek classification dataset
train_data =
pd.read_csv('C:/My_Project/Greek_dataset_1st/train.csv')
```

```
test_data =
pd.read_csv('C:/My_Project/Greek_dataset_1st/test.csv')

X_train_full = np.asarray(train_data.iloc[:, :-1])
y_train_full = np.asarray(train_data.iloc[:, -1])
X_test = np.asarray(test_data.iloc[:, :-1])
y_test = np.asarray(test_data.iloc[:, -1])

# Set the maximum value for K to 20
max_k = 20
from sklearn.model_selection import train_test_split
# Find the best K value for Greek Classification Dataset
X_train, X_val, y_train, y_val = train_test_split(X_train_full,
y_train_full, test_size=0.1, random_state=3105)
best_K_greek = find_best_K(X_train, y_train, X_val, y_val,
max_k,"Train and Validation accuracy Vs K - Greek
Classification")

# Use the best K value for Greek Classification Dataset to
calculate and print the final test accuracy
y_pred_best_greek = predict(best_K_greek, X_test, X_train_full,
y_train_full)
final_error_rate_greek, final_accuracy_greek =
accuracy_calculate(y_pred_best_greek, y_test)
print(f"\nFinal Test Accuracy with Best K ({best_K_greek}) for
Greek Classification Dataset: {final_accuracy_greek:.4f}")

# Plot error rate and accuracy vs. K for the Greek
Classification Dataset
plot_error_rate_vs_k(X_train_full, y_train_full, X_test,
y_test, max_k, "Error Rate vs. K - Greek Classification
Dataset")
# Plot accuracy vs. K for the Greek Classification Dataset
plot_accuracy_vs_k(X_train_full, y_train_full, X_test, y_test,
max_k, "Accuracy vs. K - Greek Classification Dataset")

# Plot confusion matrix for Greek Classification Dataset
plot_confusion_matrix(y_test, y_pred_best_greek,
np.unique(y_train_full), "KNN Confusion matrix for Greek
Classification Dataset")

##################################################################
#############
```

```
# # Load PCA output data
pca_output_data = pd.read_csv('C:/My_Project/pca_output.csv',
header=None)
X_pca = np.asarray(pca_output_data.iloc[:, :-1])
y_pca = np.asarray(pca_output_data.iloc[:, -1])

# # Split the PCA data into training, validation, and test sets
X_train_pca, X_val_pca, x_test_pca, y_train_pca, y_val_pca,
y_test_pca = train_val_test_split(X_pca, y_pca, val_size=0.1,
test_size=0.1, random_state=3105)

# Find the best K value for PCA output data
best_K_pca = find_best_K(X_train_pca, y_train_pca, X_val_pca,
y_val_pca, max_k,"Train and Validation accuracy Vs K - GCDB
Dataset")

# Use the best K value for PCA output data to calculate and
print the final test accuracy
y_pred_best_pca = predict(best_K_pca, x_test_pca, X_train_pca,
y_train_pca)
final_error_rate_pca, final_accuracy_pca =
accuracy_calculate(y_pred_best_pca, y_test_pca)
print(f"\nFinal Test Accuracy with Best K ({best_K_pca}) for
PCA Output Data: {final_accuracy_pca:.4f}")

plot_error_rate_vs_k(X_train_pca, y_train_pca, x_test_pca,
y_test_pca, max_k, "Error Rate vs. K - GCDB Dataset")
plot_accuracy_vs_k(X_train_pca, y_train_pca, x_test_pca,
y_test_pca, max_k, "Accuracy vs. K - GCDB Dataset")

# # Plot confusion matrix for PCA output data
plot_confusion_matrix(y_test_pca, y_pred_best_pca,
np.unique(y_pca), "KNN Confusion matrix for GCDB Dataseta")


# # Finally, the code execution ends here
```