MACHINE LEARNING (COMP7703)

Assignment - Semester 1, 2025.

Student Name: Volter Entoma

Student ID: 44782711

Machine Learning Approaches for Height Classification from Table Tennis Swing Data: A Comparative Study

Contents

1 Introduction

There are many different machine learning algorithms, and many metrics to evaluate them. In this assignment, we will focus on the K-Nearest Neighbour (K-NN) algorithm, Random Forest, and Deep Neural Networks, and the accuracy of each model. We will also look at the different metrics to evaluate the performance of each model, and use these metrics to compare the suitability of each model for the given data.

2 Aims

The primary aim of this paper is to evaluate and compare the performance of three machine learning classifiers—K-Nearest Neighbours (K-NN), Random Forest, and Deep Neural Networks (DNN)—on the TTSWING dataset for the task of classifying player height categories based on swing sensor data. The specific objectives are as follows:

- To preprocess and analyze the TTSWING dataset, addressing issues such as data imbalance and feature scaling.
- To implement and optimize K-NN, Random Forest, and DNN classifiers for the height classification task.
- To investigate the impact of dimensionality reduction (via PCA) on model performance and computational efficiency.
- To assess each model using a range of evaluation metrics, including accuracy, ROC curve, confusion matrix, precision, recall, and F1 score.
- To perform cross-validation to evaluate model generalizability and prevent overfitting.
- To compare the strengths, weaknesses, and trade-offs of each classifier in terms of predictive performance, and computational cost
- To propose potential improvements and future directions for each classification strategy.

3 Data

The data used in this assignment is the `TTSWING` dataset, which is the provided dataset for this assignment. The dataset contains the 34 features, which are the 34 different measurements of table tennis swings. The dataset also contains the height of the player, which is the target variable. The dataset is a CSV file, and can be read into Python using the `pandas` library. The dataset is split into three categories: high, medium, and low.

3.1 Data split

The data is split into three height categories: high, medium, and low. The data is split into 60% training data, 20% testing data, and 20% validation data. The training data is used to train the each model, the testing data is used to test the accuracy of each model, and the validation data is used to compare the performance of the models.

3.2 Data Imbalance

After splitting the data, there is an imbalance in the amount of training data assigned to each category in height. There are 17364 (29.73%) data points in the high, 40800 (42.03%) data points in the medium, and 27450 (28.23%) in the low category. This is a significant imbalance; this imbalance is problematic because many machine learning algorithms tend to be biased toward the majority class. This can result in poor predictive performance for the minority classes, as the model may learn to ignore them in favor of achieving higher overall accuracy. In imbalanced datasets, accuracy becomes a misleading metric, since a model can achieve high accuracy by simply predicting the majority class most of the time, while failing to correctly classify minority class instances. This is especially concerning when the minority classes are of particular interest or importance.

The data imbalanced is addressed by the use of Synthetic Minority Over-sampling Technique (SMOTE) to generate synthetic data points for the low category. The SMOTE algorithm generates synthetic data points by interpolating between existing data points in the low category. This is done by selecting a random data point from the low category, and then selecting a random data point from the k nearest neighbours of that data point. The synthetic data point is then generated by taking a weighted average of the two data points.

After applying SMOTE, the training data is balanced, with 24552 data points in each category. All subsequent analysis is performed on the balanced training data. The testing and validation data is not balanced, and is used to test the accuracy of the models.

3.3 Data Standardisation

The data is standardized using the StandardScaler from the `sklearn` library. The StandardScaler standardizes the data by removing the mean and scaling to unit variance. This is done to ensure that all features have the same scale, and to improve the performance of the models.

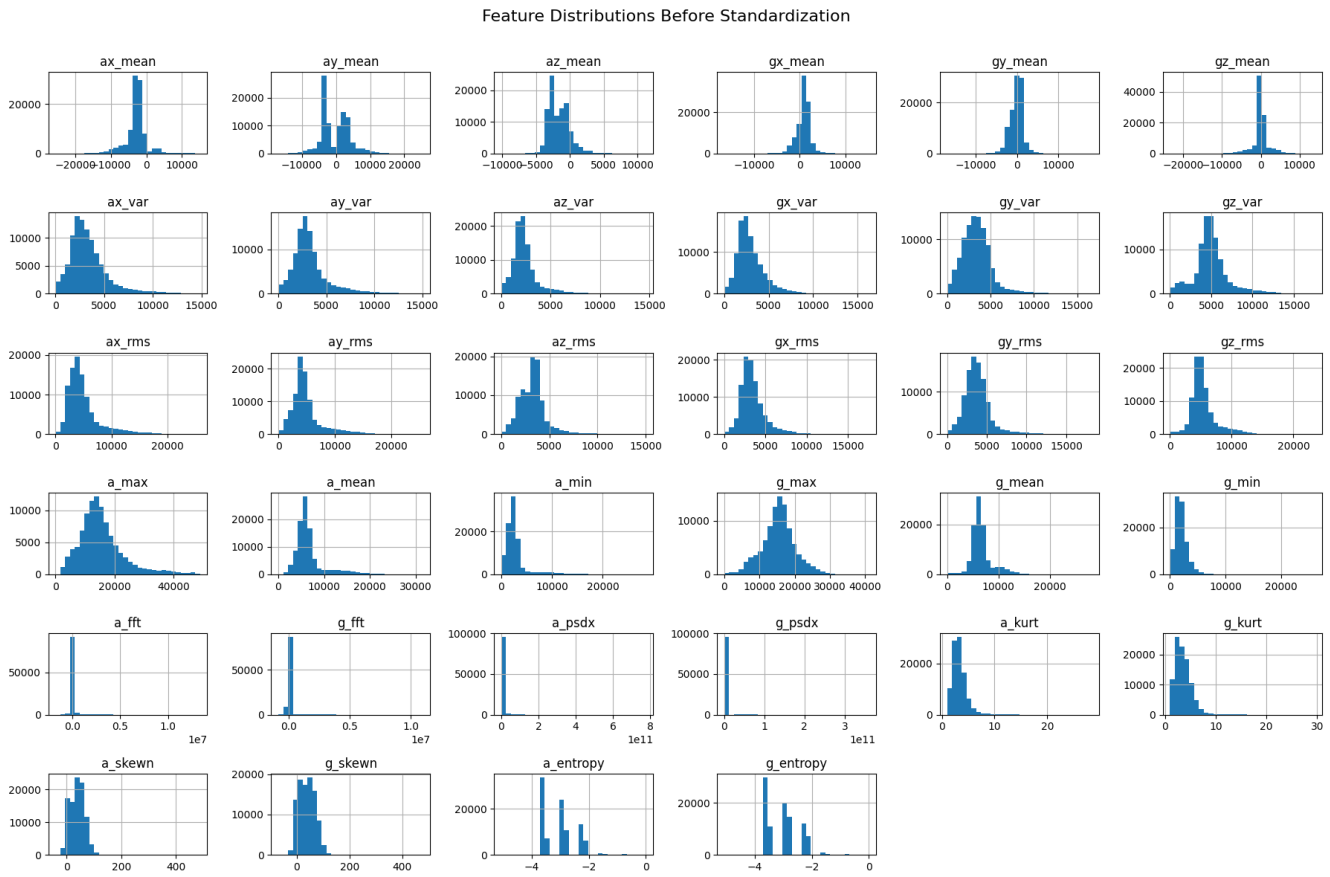

Figure 1: Distribution of the data before standardization.

4 Principle Component Analysis

To investigate the effect of dimensionality reduction on the accuracy of the models, we will use Principle Component Analysis (PCA) to reduce the dimensionality of the data. PCA is a linear transformation that transforms the data into a new coordinate system, where the first coordinate is the direction of maximum variance, the second coordinate is the direction of maximum variance orthogonal to the first coordinate, and so on.

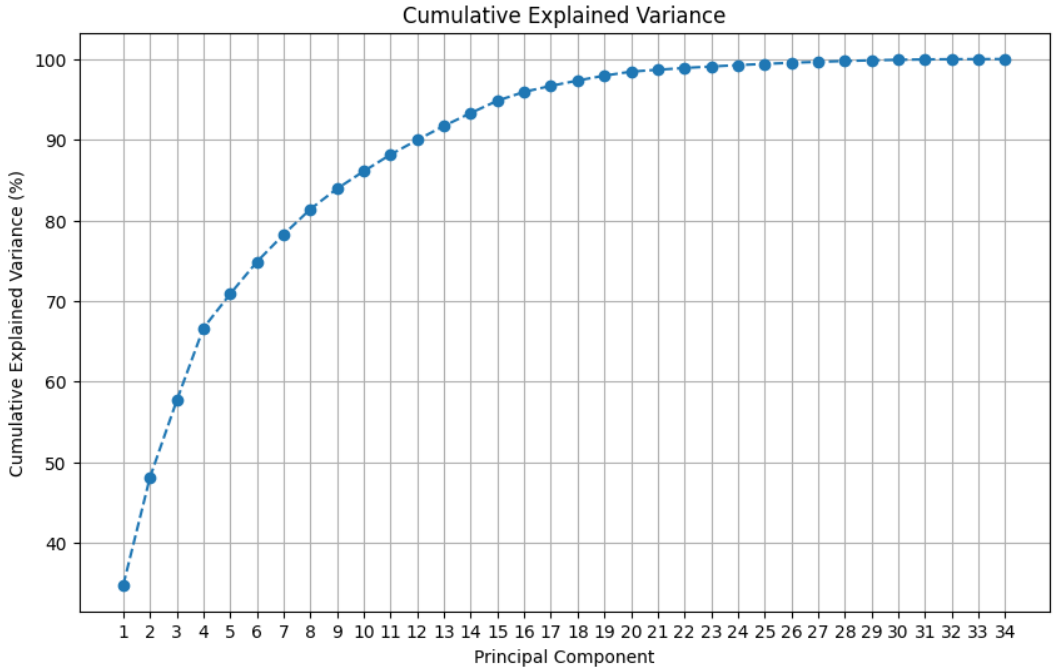

Figure 2: Cumulative variance explained by each principal component.

As shown in Figure 2, approximately 95% of the total variance is explained by the top 16 principal components. This allows us to reduce the dimensionality of our data from 34 dimensions to 16 dimensions while retaining the vast majority of the information content.

By taking a look at the correlation matrix of the data, as shown in Figure 3, we can see that there are many features that are highly correlated with each other. These off-diagonal correlations could explain why the PCA transformation is able to reduce the dimensionality of the data while retaining a high percentage of the variance.

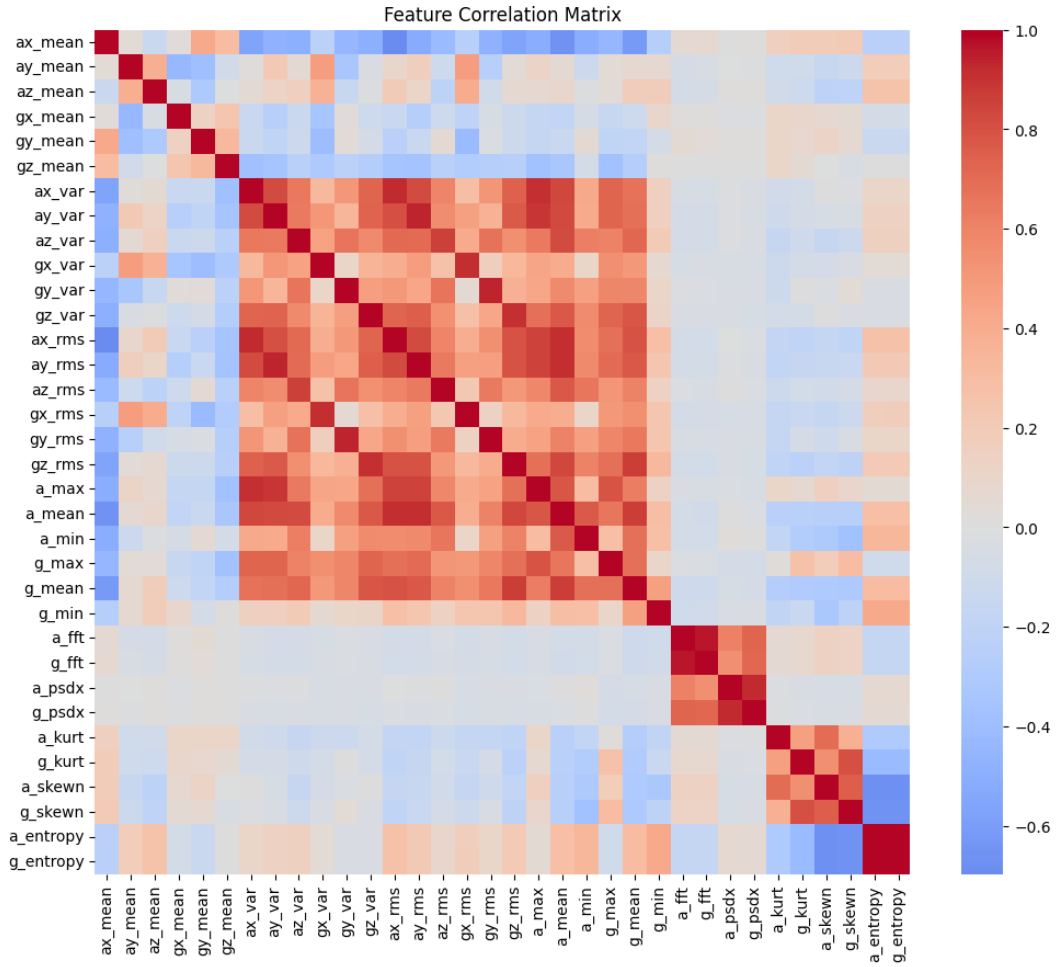

Figure 3: Correlation matrix of the standardized data.

Dimensional reduction offers several benefits:

- Reduced computational complexity in subsequent modeling
- Mitigation of the curse of dimensionality
- Removal of potentially noisy or redundant features
- Improved model interpretability

The effectiveness of this dimensionality reduction will be evaluated by comparing model performance on both the original and PCA-reduced datasets.

5 K-NN Classifier

The K-Nearest Neighbour (K-NN) algorithm is a simple and effective machine learning algorithm that can be used for both classification and regression tasks. The K-NN algorithm works by finding the k nearest neighbours of a data point, and then predicting the class of the data point based on the classes

of the k nearest neighbours. The K-NN algorithm is a non-parametric algorithm, which means that it does not make any assumptions about the distribution of the data. This makes the K-NN algorithm very flexible, and it can be used for a wide variety of tasks, including for our case of the classifying the height of a table tennis player based on their swing data.
The K-NN algorithm was applied to the data to classify the height of the player based on their swing data.

5.1 Accuracy of K-NN Classifier

Using the data split outlined in Section 3.1, the K-NN classifier was trained on the training data, and then tested on the testing data.

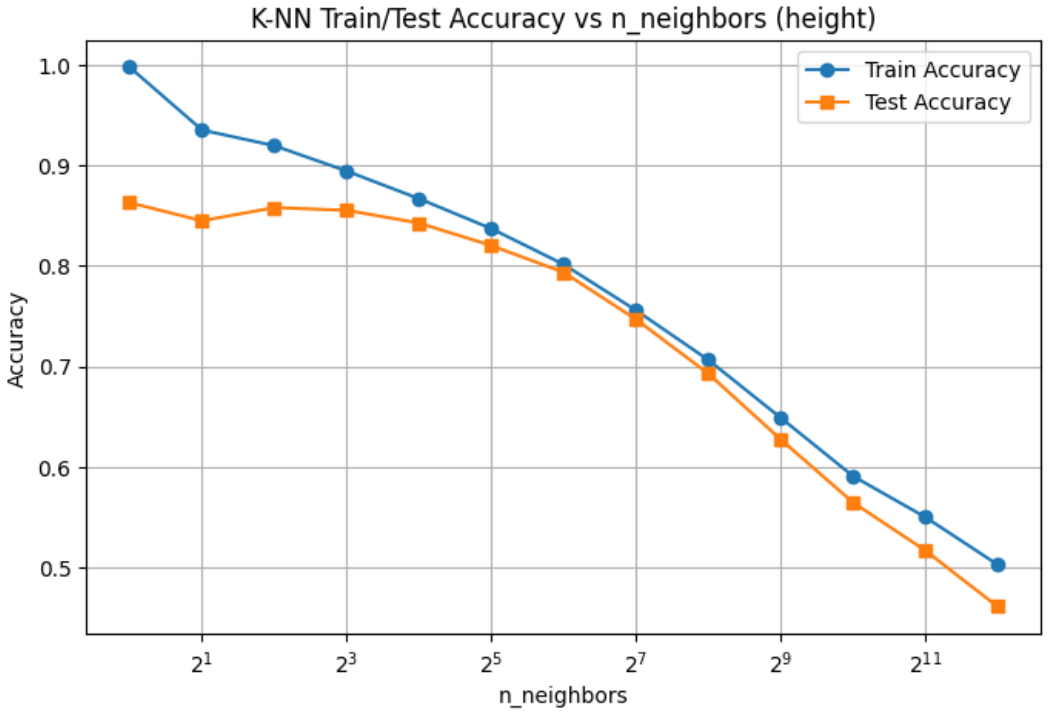

Figure 4: Accuracy of K-NN classifier against the train and test data.

The accuracy of the K-NN classifier is heavily dependent on the number of neighbours (k) used in the algorithm. Figure 4 shows the accuracy of the K-NN classifier reaches a maximum at 1 neighbour, and then decreases as the number of neighbours increases. This could be related to the high dimensionality of the data.
The K-NN classifier is particularily sensitive to the curse of dimensionality; as the number of dimensions increases, point within the hyperspace are more likely to be equidistant from each other. This means that the K-NN algorithm is less able to distinguish between points, and hence the accuracy of the K-NN classifier decreases. This is a common problem with high dimensional data, and is one of the reasons why dimensionality reduction techniques such as PCA are used.

We can confirm this by looking at the accuracy of the K-NN classifier on the PCA-reduced data.

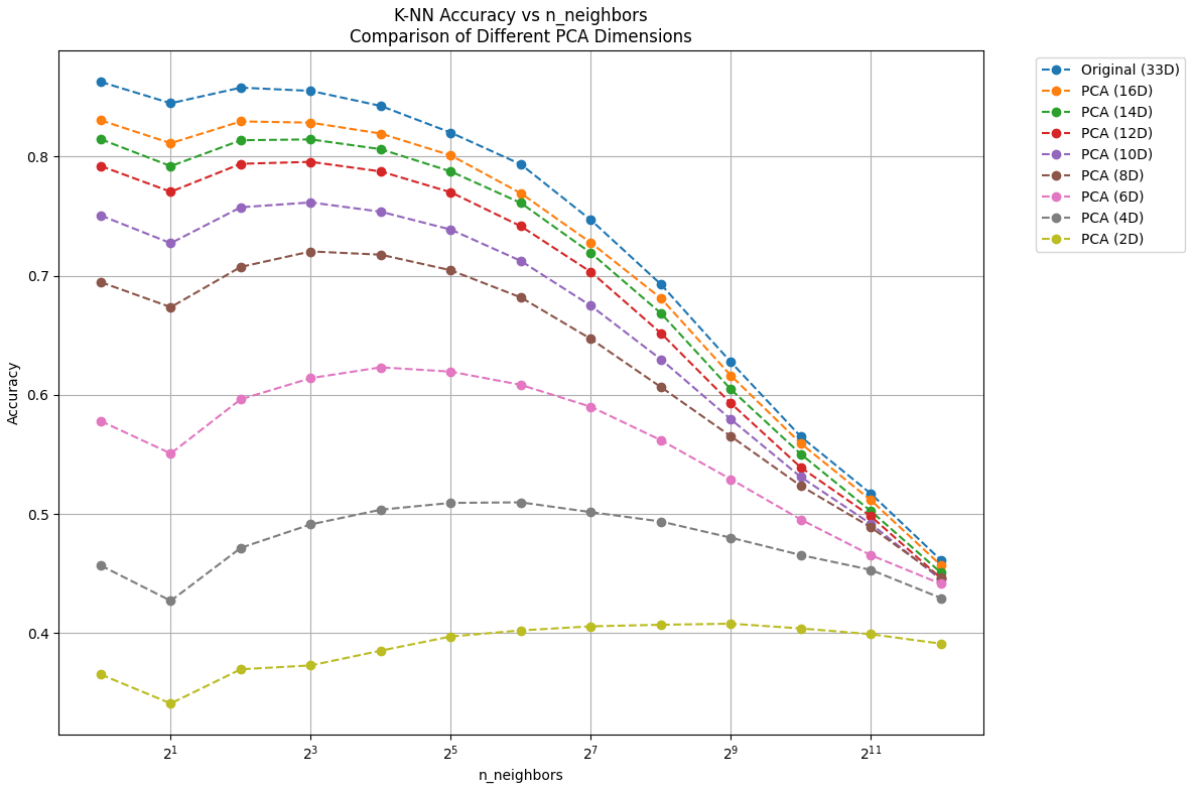

Figure 5: Accuracy of K-NN classifier on PCA-reduced data against the test data.

The accuracy of the K-NN classifier on the PCA-reduced data is shown in Figure 5. The curve for reduced data to 2-dimensions shows the best accuracy occurs with 2^9 neighbour, which suggests that the points become more "spread-out" in 2-dimensions and the model is able to distinguish between different categories better as the number of neighbours increases.
However, despite the dimensionality reduction, the accuracy of the K-NN classifier is still highest at 1 neighbour, and then decreases as the number of neighbours increases. This suggests that the decrease in accuracy as neighbours increase is not solely due to the high dimensionality of the data, but also due to the nature of the K-NN algorithm on this particular dataset. This idea is supported by the fact that the curve for the PCA-reduced data (to 16 dimensions) is similar to the curve for the original data.
Additionally, the accuracy of the K-NN classifier decreases as the number of dimensions decreases. If the curse of dimensionality was a significant problem for this dataset, we would expect the accuracy to increase. This suggests that the curse of dimensionality is not a significant problem for this dataset, and that decreasing the number dimensions is not beneficial to the model, as it loses too much information.

5.2 K-Fold Cross-validation

The best accuracy of the K-NN classifier is achieved with one neighbour, and the accuracy is approximately 86.26%. However, one neighbour could be overfitting the data, therefore is important to explore different sets of training and testing data, to see if the accuracy is consistent across different sets of data. This is done by using K-Fold Cross-validation, which splits the data into K different sets, and then trains and tests the model on each set. The accuracy of the model is then averaged over all K sets.
We use 10-fold cross-validation, which means that the data is split into 10 different sets, and then the

model is trained and tested on each set. Note that the choice of the number of folds was arbitrary, and was chosen to be 10 because it is a common choice in the literature.

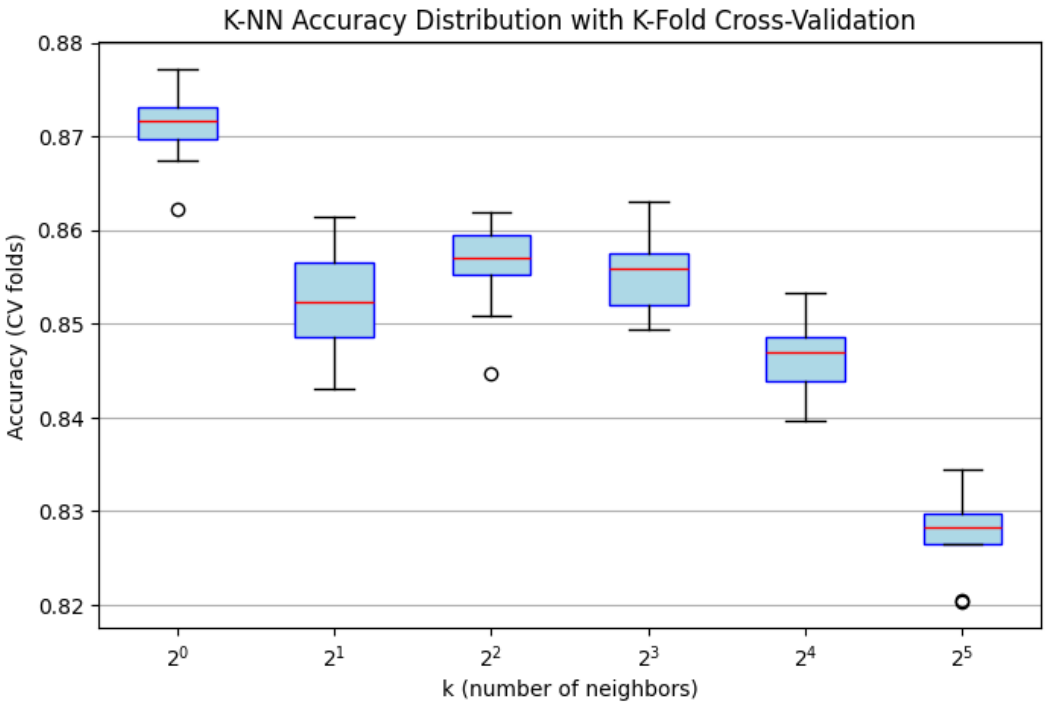

Figure 6: Accuracy of K-NN classifier with K-Fold Cross-validation. Each box and whisker plots shows the spread of the accuracy against the test data across the 10 folds. The red line shows the mean accuracy across the 10 folds.

The accuracy of the K-NN classifier with K-Fold cross-validation is shown in Figure 6. With 10-fold cross-validation, the highest average accuracy occurs with one neighbour. Notably, the minimum accuracy across the 10 folds for one neighbour is approximately equal to the maximum accuracy observed for two, four, and eight neighbours. This suggests that one neighbour is indeed the optimal choice for this dataset.

Additionally, the variance in accuracy across folds for one neighbour is comparable to the variance observed for other values of k. If one neighbour were overfitting the data, we would expect to see significantly higher variance across the folds for k=1 compared to other values of k. Since this is not the case, we can conclude that, for this dataset, the K-NN classifier with one neighbour does not appear to overfit, as evidenced by the consistently low variance.

5.3 ROC Curve

We use the k=1 K-NN model with the highest accuracy in the 10-fold cross-validation to plot the ROC curve. The ROC curve is a graphical representation of the performance of a binary classifier as the discrimination threshold is varied. It plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The area under the ROC curve (AUC) is a single scalar value that summarizes the performance of the classifier across all thresholds.

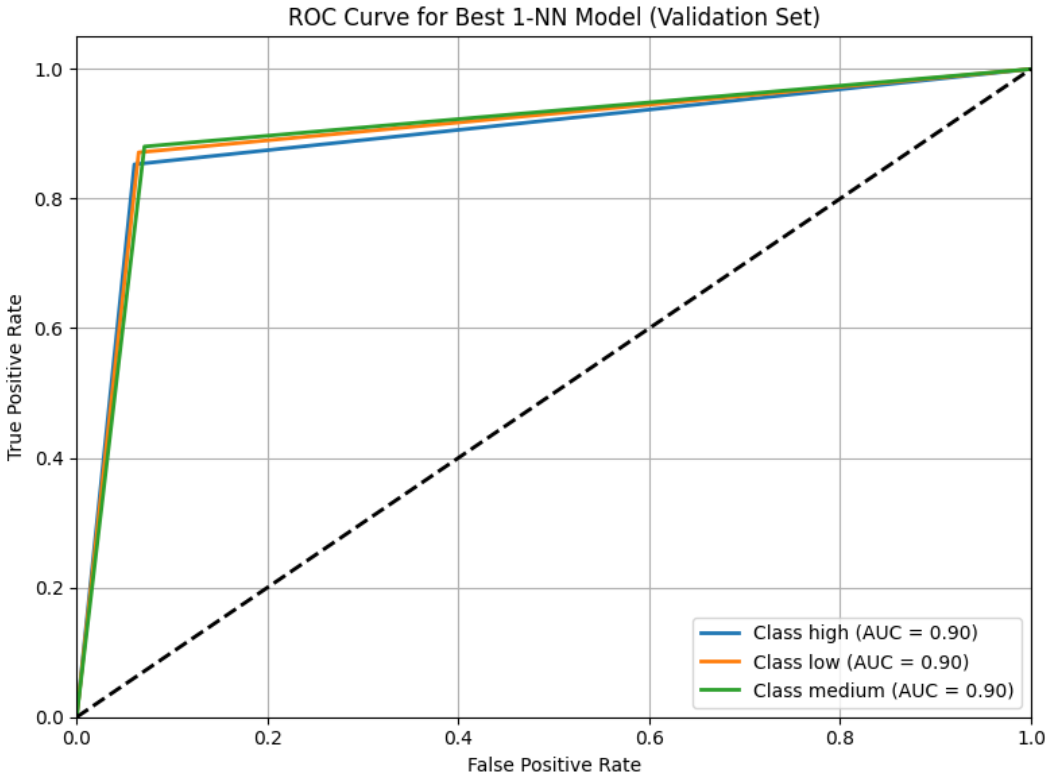

Figure 7: ROC curve for K-NN classifier with k=1.

The ROC curve for the K-NN classifier with k=1 is shown in Figure 7. The AUC for the K-NN classifier with k=1 is approximately 0.9 for all height categories. This indicates that the K-NN classifier with k=1 is a good classifier, as it has a high true positive rate and a low false positive rate.
The ROC curve itself is not very informative for one-neighbour K-NN, as it can only produce three points. The lack of points is due to the fact that the model outputs "hard" probabilities (only 0 or 1) for each class, rather than "soft" probabilities, that would be possible for a higher number of neighbours.

5.4 Confusion Matrix

We can also use the same K-NN model to plot the confusion matrix using the validation data that was not used in the training or testing of the model. The confusion matrix is a table that is often used to describe the performance of a classification model on a set of data for which the true values are known. The confusion matrix shows the number of correct and incorrect predictions made by the model, broken down by class.

The confusion matrix shows that the diagonal elements to be the largest, which indicates that the K-NN classifier with k=1 is able to correctly classify the majority of the data points.
The off-diagonal elements are much smaller, which indicates that the K-NN classifier with k=1 is not making many incorrect predictions. This is a good sign, as it indicates that the K-NN classifier with k=1 is a good classifier for this dataset.
The validation set has a significant imbalance in the number of data points in each category; similar to the raw training data, the medium category has the most data points, followed by the low and high categories. However, SMOTE was not applied to validation because artificially generating data points would

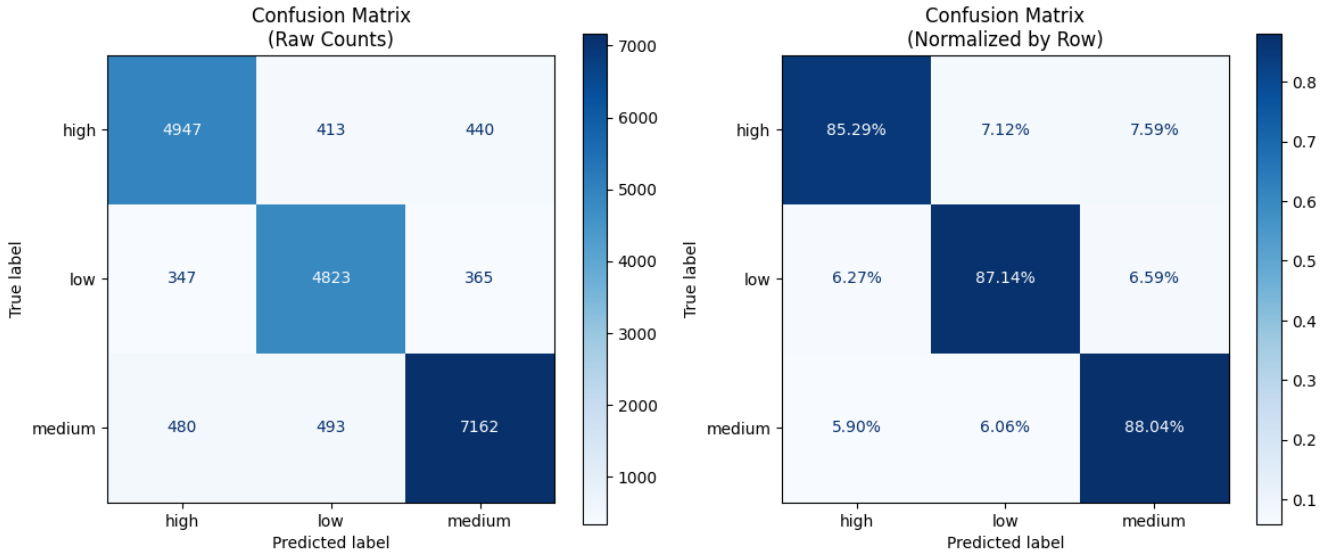

Figure 8: Confusion matrix for K-NN classifier with k=1. (Left) Confusion matrix against the raw validation set. (Right) Confusion matrix normalized by row.

give misleading performance metrics. Instead, the confusion matrix is normalized by row, which allows us to see the proportion of correct and incorrect predictions for each category. The normalized confusion matrix shows that the K-NN classifier with k=1 is able to correctly classify the majority of the data points in each category.

5.5 Precision, Recall, and F1 score

The precision, recall, and F1 score are three metrics that are commonly used to evaluate the performance of a classification model. The precision is the number of true positive predictions divided by the total number of positive predictions. The recall is the number of true positive predictions divided by the total number of actual positive instances. The F1 score is the harmonic mean of precision and recall.

Table 1: Precision, Recall, and F1 Score for each class (Validation Set)

Class	Precision	Recall	F1-score
high	0.86	0.85	0.85
low	0.84	0.87	0.86
medium	0.90	0.88	0.89

All three classes have a high precision, recall, and F1 score, which indicates that the K-NN classifier with k=1 is able to correctly classify the majority of the data points in each category. Additionally, the scores are relatively balanced across all classes, which suggests that the K-NN classifier with k=1 is not biased towards any particular class, despite the imbalanced validation data.

6 Random Forest Classifier

The Random Forest classifier is an ensemble learning method that constructs a multitude of decision trees during training and outputs the mode of the classes (classification) or mean prediction (regression) of the individual trees. It is particularly effective for high-dimensional datasets and can handle both categorical and continuous variables. The Random Forest algorithm is robust to overfitting, especially when the number of trees in the forest is large. In this section, we will explore the performance of the Random Forest classifier on the TTSWING dataset.
Random Forest builds many decision trees using random subsets of the data and features, and aggregates their predictions. The quality of each split in a tree is measured using criteria such as Gini impurity (for classification), Entropy (information gain), or Mean Squared Error (for regression).
To keep this report within reasonable constraints, we will only explore the Gini impurity, and explore other parameters within the Random Forest classifier. The Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset.

6.1 Accuracy of Random Forest Classifier

The Random Forest classifier was trained on the balanced and scaled training data, and then tested on the testing data. Various values of the number of trees in the forest (10, 50, 100, 200, 500, 800, and 1400), and the maximum depth of the trees (5, 10, 20, 30, 40, and no limit) were explored. The accuracy of the Random Forest classifier was then calculated for each combination of parameters.

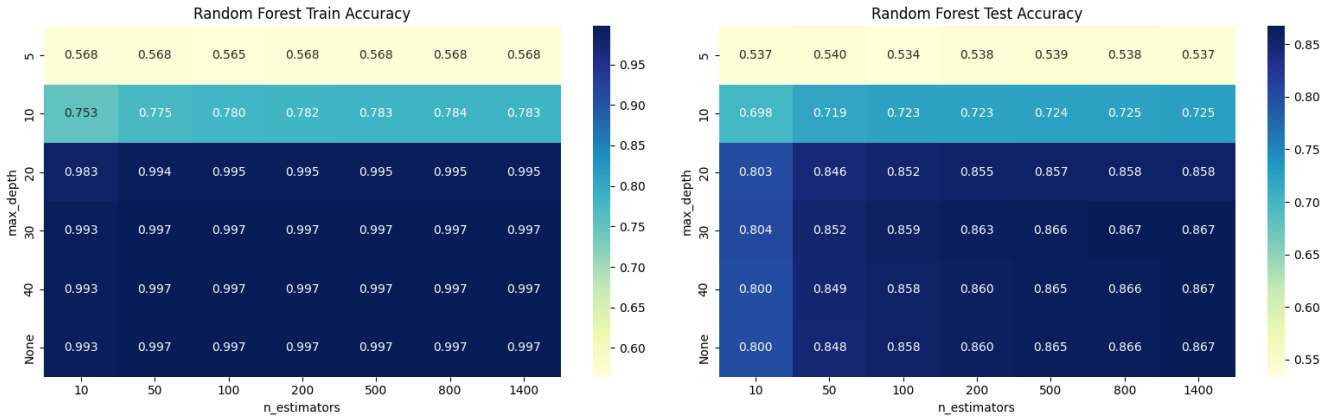

Figure 9: Accuracy of Random Forest classifier against the train and test data.

We can see from Figure 9 that the accuracy of the Random Forest classifier increases as the number of trees in the forest increases, and as the maximum depth of the trees increases. This is expected, as more trees and deeper trees allow for more complex decision boundaries to be learned. However, there is a point of diminishing returns, where increasing the number of trees or depth does not significantly improve accuracy.
As a caveat, the accuracy of the Random Forest classifier is not guaranteed to increase as the maximum depth of the trees increases. This is because deeper trees are more likely to overfit the data, and hence the accuracy may decrease as the maximum depth increases. This is particularly evident in all trees that were explored - the accuracy of the Random Forest classifier peaks at a maximum depth of either 20 or 30,

then decreases slightly as the maximum depth increases. This suggests that the Random Forest classifier is able to learn complex decision boundaries without overfitting the data, but that there is a limit to how complex the decision boundaries can be before overfitting occurs.

6.1.1 Computational time after dimensionality reduction

A big drawback with increasing the number of trees and the maximum depth of the trees is the computational time. The computational time of the Random Forest classifier is significantly increased as the number of trees and maximum depth increases. This is because each tree in the forest must be trained on the data, and each tree must be traversed to make a prediction. This can be mitigated by using dimensionality reduction techniques such as PCA, which reduces the number of features in the data, and hence reduces the computational time.

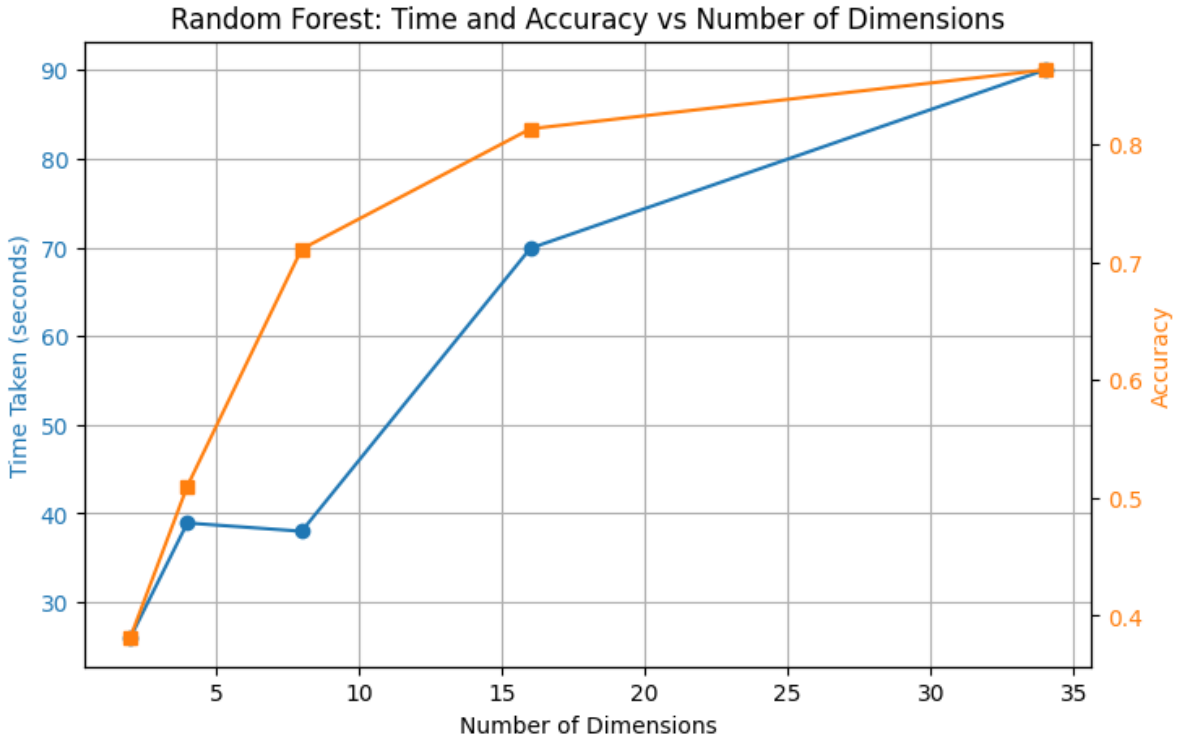

Figure 10: Accuracy of Random Forest classifier on PCA-reduced data. The model parameters were set to 200 trees and a maximum depth of 30.

When the data is reduced to 16-dimensions using PCA, the computational time of the Random Forest classifier significantly decreases. However, the accuracy of the Random Forest classifier slightly decreases as well. This is expected, as the PCA-reduced data loses some information content, and hence the accuracy of the Random Forest classifier decreases. However, the decrease in accuracy is not significant, and the computational time is significantly reduced. This is a strong argument for using PCA to reduce computational cost, without significantly sacrificing accuracy. However, since the time taken to train the Random Forest classifier on the original data is within reason, and the accuracy is slightly higher, we will use the original data for the rest of the analysis.

6.2 K-Fold Cross-validation

Increasing the number of trees does not increase risk of overfitting, however, increasing the maximum depth of the trees does increase the risk of overfitting. We can take advantage of this by deciding the best number of trees, since we are not limited by the number of trees with regards to overfitting. The accuracy of the Random Forest classifier only marginally increases after 100 trees, and the computational time increases exponentially after 100 trees. Therefore, we will use 100 trees and vary the maximum depth of the trees to investigate the effect of overfitting on the accuracy of the Random Forest classifier.

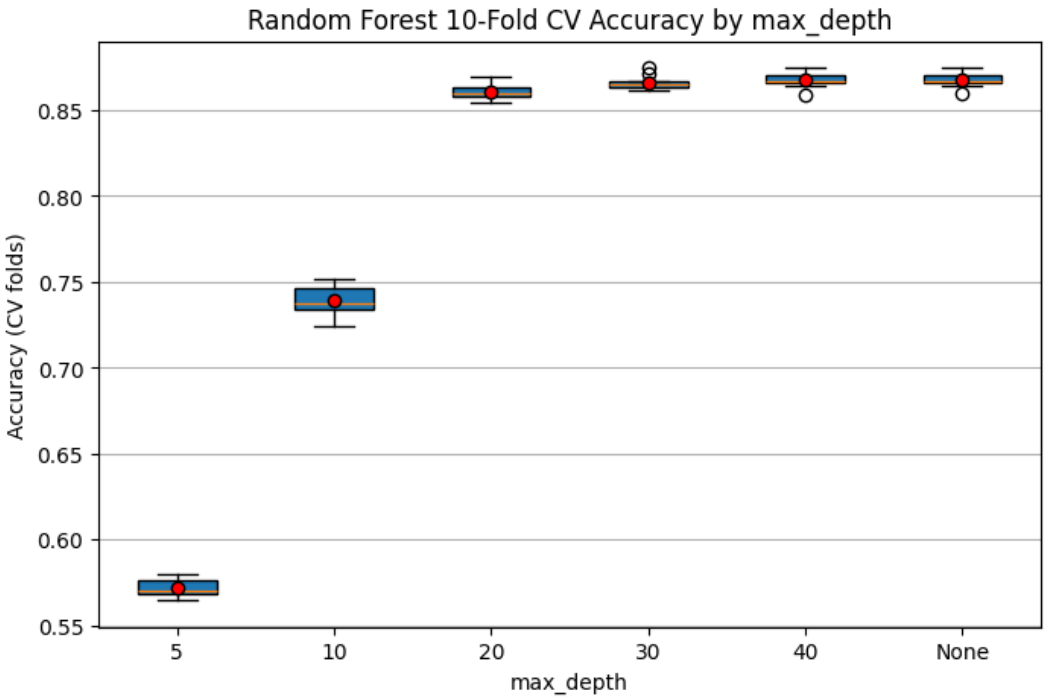

Figure 11: Accuracy of Random Forest classifier with K-Fold Cross-validation. Each box and whisker plots shows the spread of the accuracy against the test data across the 10 folds. The red line shows the mean accuracy across the 10 folds.

The distribution of the accuracy shows a low variance across all maximum depths, which suggests that the Random Forest classifier is not overfitting the data. The maximum depth of None (unlimited) achieved the highest mean accuracy, while still maintaining a low variance, therefore we will use this model as the standard model for the Random Forest classifier.
Note that the means of the accuracies for maximum depths larger than 20 are very similar. Additionally, the mean accuracy for 40 is within the IQR of the maximum depth of None. It is fair to assume the true mean across the maximum depths larger than 20 are insignificantly different, and so the choice of maximum depth greater than 20 is arbitrary, in terms of accuracy. Despite the fact that the computational time is significantly increased as the maximum depth increases, we will use the maximum depth of None, because the time taken is within reason, and the accuracy, for this dataset, is the highest. The Random Forest classifier with 100 trees and unlimited depth achieved a mean accuracy of approximately 86.75%, which is slightly higher than the K-NN classifier with k=1, which achieved a mean accuracy of approximately 86.26%. This suggests that the Random Forest classifier is a better classifier for this dataset than the K-NN classifier, and that the Random Forest classifier is able to learn more complex decision boundaries than the K-NN classifier.

6.3 ROC curve

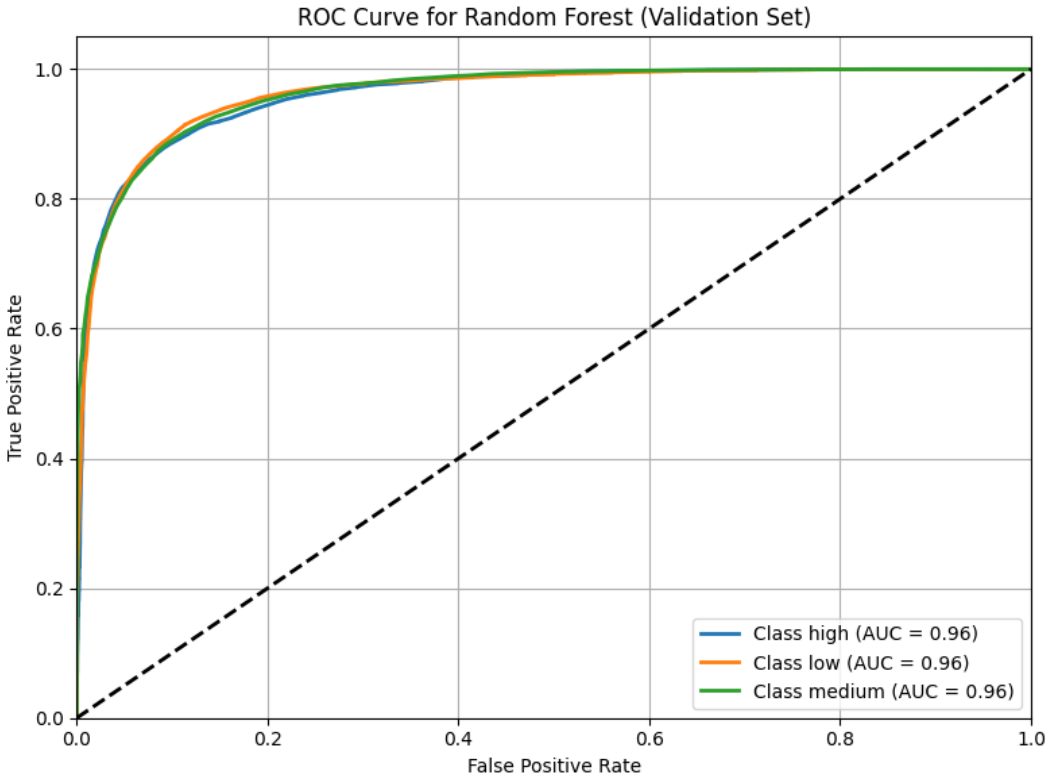

Figure 12: ROC curve for Random Forest classifier with 100 trees and maximum depth of None.

The ROC curve for the Random Forest classifier with 100 trees and unlimited depth is shown in Figure 12. The area under the curve (AUC) is approximately 0.96 for all height categories, indicating strong discriminative performance with a high true positive rate and a low false positive rate across classes. Notably, the Random Forest achieves a higher AUC than the K-NN classifier with (k=1), suggesting superior classification performance on this dataset.
This improvement is consistent with the strengths of ensemble methods like Random Forest, which aggregate the predictions of multiple decision trees to reduce variance and improve generalization. Additionally, Random Forests are less prone to overfitting compared to single decision trees, particularly when a sufficient number of trees are used.

6.4 Confusion matrix

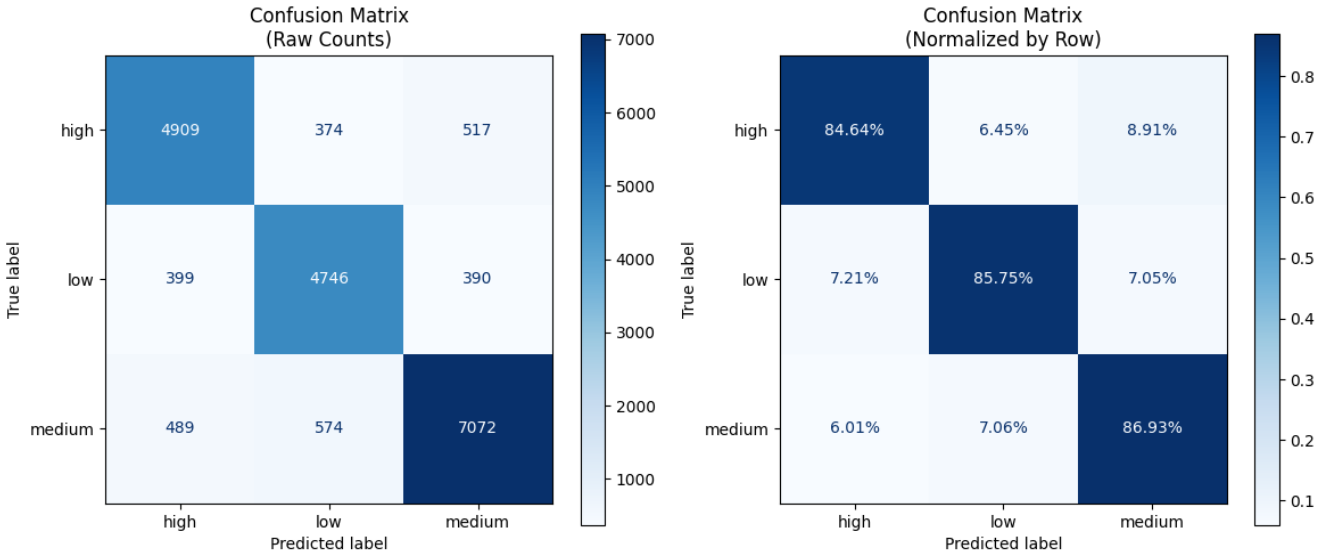

Figure 13: Confusion matrix for Random Forest classifier with 100 trees and maximum depth of None. (Left) Confusion matrix against the raw validation set. (Right) Confusion matrix normalized by row.

Similar to the K-NN classifier, the diagonal elements are the largest, which indicates that the Random Forest classifier with 100 trees and unlimited depth is able to correctly classify the majority of the data points.
The diagonal elements for the Random Forest classifier are slightly lower than the K-NN classifier, which indicates that the chosen Random Forest classifier is only slightly less accurate than the K-NN classifier.

6.5 Precision, Recall, and F1 score

Class	Precision	Recall	F1-score
high	0.85	0.85	0.85
low	0.83	0.86	0.85
medium	0.89	0.87	0.88

Table 2: Precision, recall, and F1-score for each class on the validation set.

The precision, recall, and F1 score are very similar to that of the K-NN classifier and are all high for all three classes, which indicates that the Random Forest classifier with 100 trees and unlimited depth is able to correctly classify the majority of the data points in each category. Additionally, the scores are relatively balanced across all classes, which suggests that the Random Forest classifier with 100 trees and unlimited depth is not biased towards any particular class.

7 Deep Neural Networks

The Deep Neural Network (DNN) is a powerful machine learning algorithm that can be used for both classification and regression tasks. The DNN algorithm works by using multiple layers of neurons to learn complex decision boundaries. For this section, we will explore two-hidden layer DNNs, which are a type of feedforward neural network. The model is trained using the Cross Entropy loss function, with a L2 regularization term to prevent overfitting. The model is trained using the Adam optimizer, which is a popular optimization algorithm for training neural networks. The model is trained on the scaled training data, and then tested on the testing data.

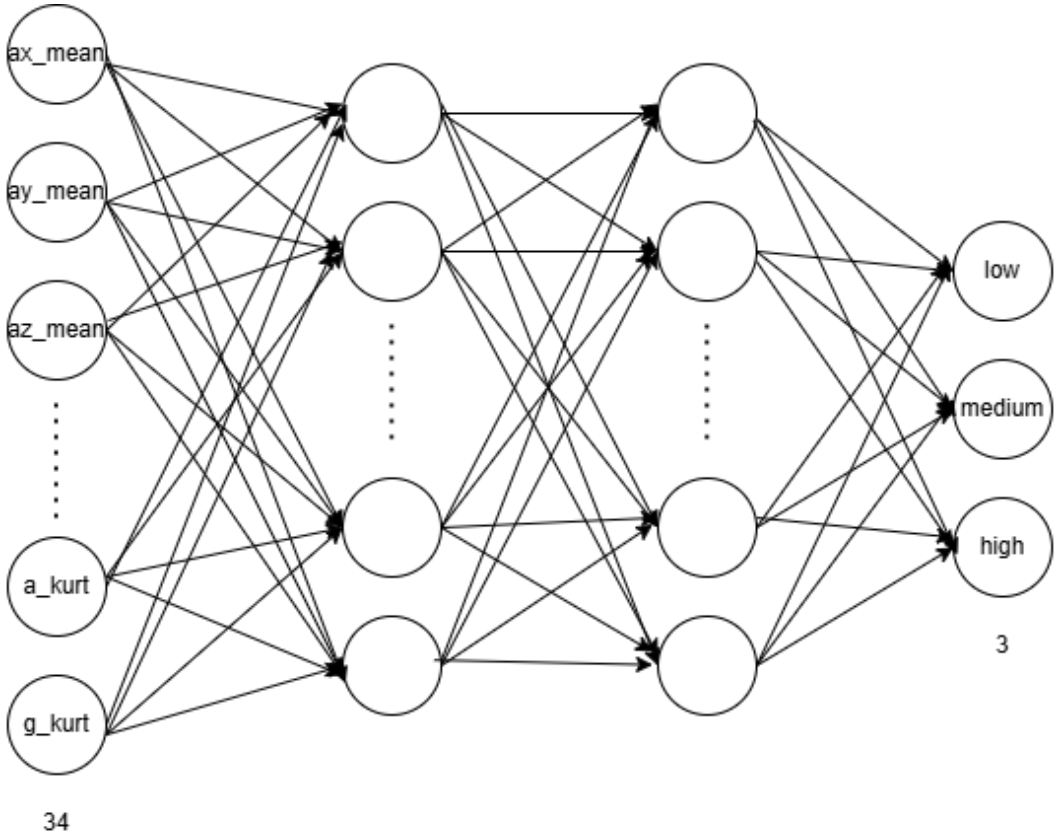

Figure 14: Architecture of the DNN with two hidden layers. The input layer has 34 nodes. There are two hidden layers with a variable number of nodes. The output layer has 3 nodes, corresponding to the three height categories.

7.1 Training

Initial assessment of the DNN were performed using a 128 nodes in each hidden layers. It was found that the DNN can achieved a accuracy on-par with the previous models when trained using a batch size of 16, learning rate of 0.001, and 100 epochs. However, it is unclear if the model is overfitting, and so we will use 5-fold cross-validation to investigate the effect of the different hidden layer sizes on the accuracy of the DNN.

7.1.1 Effect of hidden layer sizes

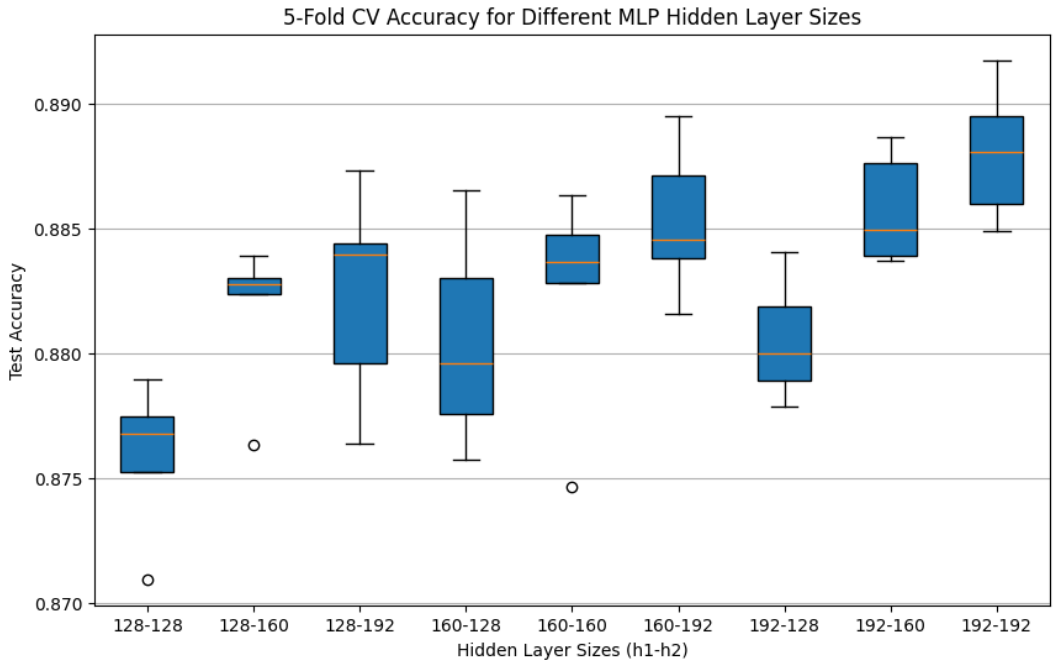

Figure 15: Different hidden layer sizes (h1-h2) with 5-fold cross-validation. Each box and whisker plots shows the spread of the accuracy against the test data across the 5 folds. The red line shows the mean accuracy across the 5 folds.

There appears to be a trend for increasing the size of the hiddens layers to increase the accuracy of the DNN. Notably, the biggest increase in accuracies occurs when the second hidden layer is increased, for the same size of the first hidden layer. This suggests that the second hidden layer is more important than the first hidden layer, and that increasing the size of the second hidden layer will increase the accuracy of the DNN.
The DNN with 192-192 hidden layers has the highest mean accuracy, and has a low variance, that is comparable to the other models. This suggest that the DNN with 192-192 hidden layers is a good choice for this dataset, and that the DNN is not overfitting the data, or at least not overfitting the data more than the other models.

7.1.2 Training 192-192 DNN model

The best model structure from the 5-fold cross-validation is the DNN with 192-192 hidden layers, which has a mean accuracy of approximately 88.8%. We use this model as the standard model for the DNN, and train it on the full training data. The model is trained using a batch size of 16 and an initial learning rate of 0.001, and 100 epochs. The model was paused every 20 epochs to fine-tune the learning rate, and the model was trained for a total 116 epochs. After the learning rate was lowered, 100 epochs were trained, there was no significant increase the in the train and test accuracy, so the model was stopped at 116 epochs. The learning rate was lowered during the training process to prevent overshooting the minimum of the loss function, while still training the model at a reasonable speed. The train/test curve are shown in Figure 16.

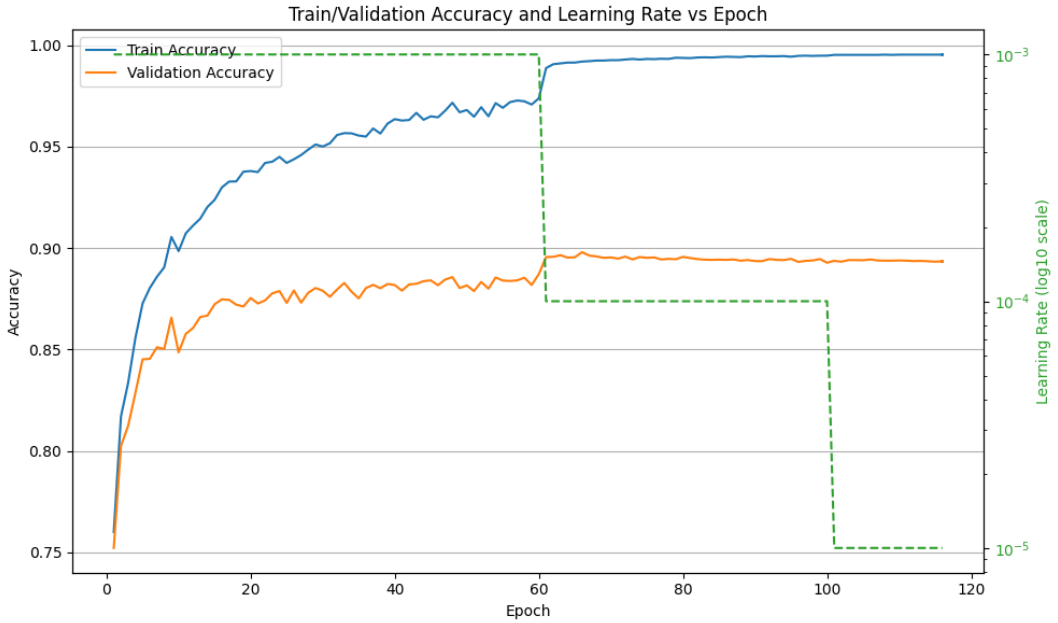

Figure 16: Train/test curve for the DNN with 192-192 hidden layers. The blue line shows the training accuracy, and the orange line shows the testing accuracy. The learning rate was lowered whenever the testing accuracy did not increase for 20 epochs. The model was trained for a total of 116 epochs.

7.2 ROC Curve

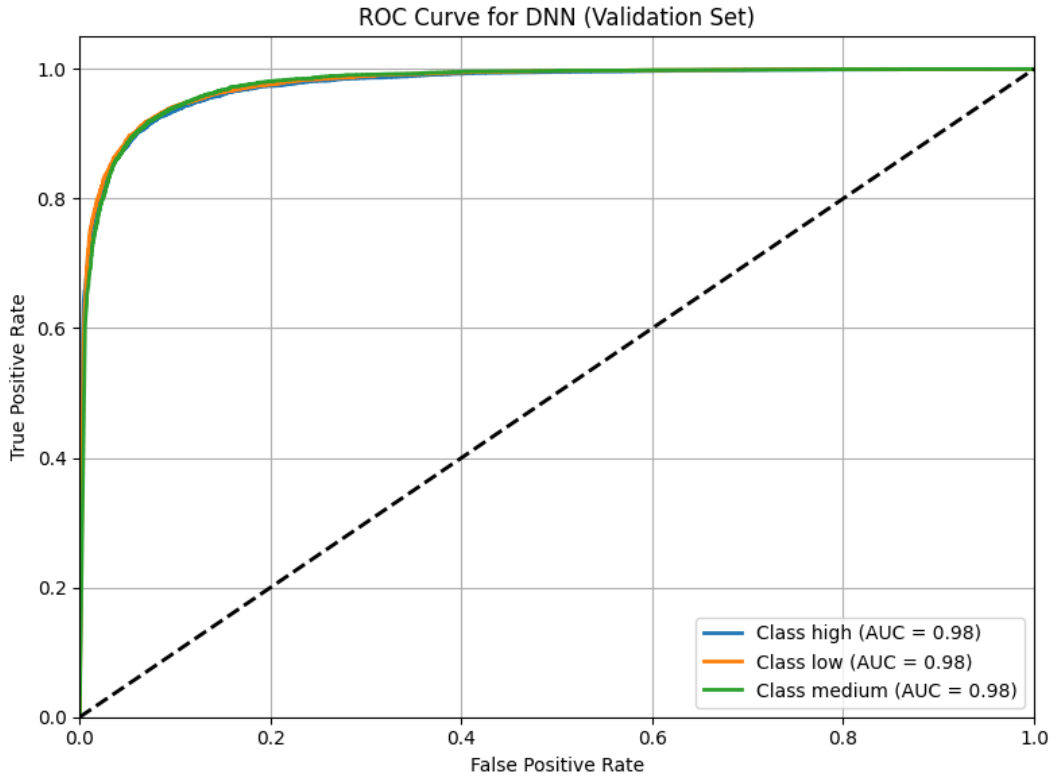

Figure 17: ROC curve for DNN with 192-192 hidden layers.

The area under the curve (AUC) is approximately 0.98 for all height categories, indicating strong discriminative performance with a high true positive rate and a low false positive rate across classes. The AUC is slightly higher than that of the Random Forest classifier, which suggests that the DNN is a better classifier for this dataset. This is expected, as the DNN is able to learn more complex decision boundaries than the Random Forest classifier, due to the multiple layers of neurons.

7.3 Confusion Matrix

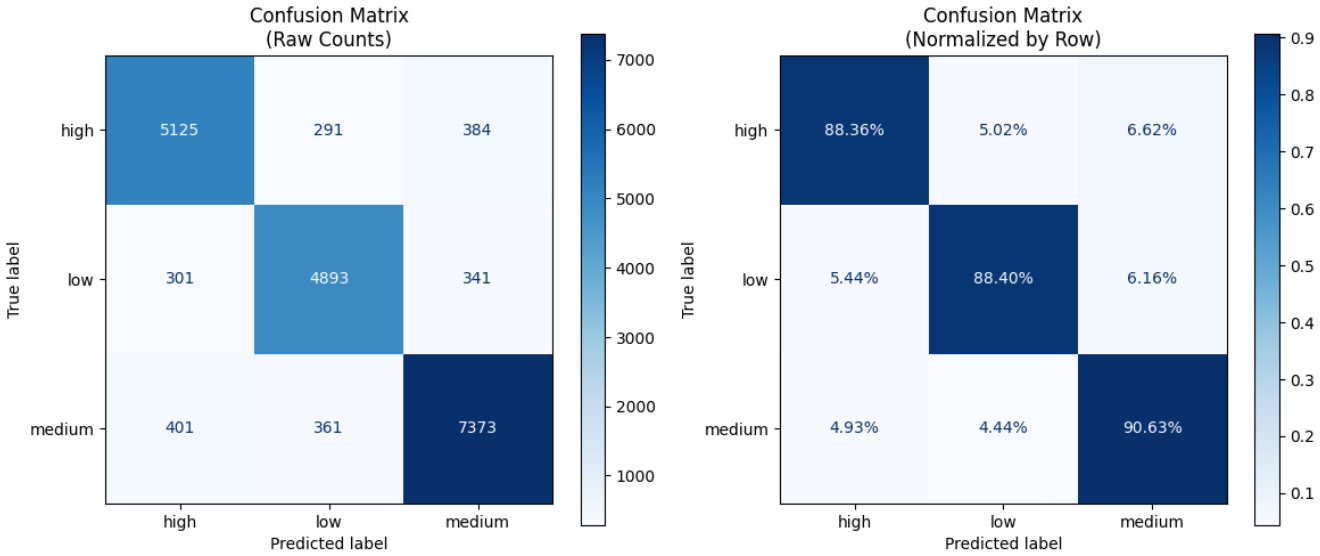

Figure 18: Confusion matrix for DNN with 192-192 hidden layers. (Left) Confusion matrix against the raw validation set. (Right) Confusion matrix normalized by row.

Similar to the K-NN and Random Forest classifiers, the diagonal elements are the largest, which indicates that the DNN with 192-192 hidden layers is able to correctly classify the majority of the data points. The diagonal elements for the DNN are slightly higher for all three classes, compared to the K-NN and Random Forest classifiers, which indicates that the DNN with 192-192 hidden layers is able to correctly classify more data points than the other two classifiers. This corroborates with the higher AUC of the DNN, and suggests that the DNN is a better classifier for this dataset.

7.4 Precision, Recall, and F1 score

Class	Precision	Recall	F1-score
high	0.88	0.88	0.88
low	0.88	0.88	0.88
medium	0.91	0.91	0.91

Table 3: Precision, Recall, and F1-score for each class on the validation set using the trained DNN model.

The precision, recall, and F1 scores are all high for all three classes, which indicates that the DNN with 192-192 hidden layers is able to correctly classify the majority of the data points in each category.

Additionally, the scores are relatively balanced across all classes, which suggests that the DNN with 192-192 hidden layers is not biased towards any particular class.

8 Conclusion

This report presented an analysis of the TTSWING dataset using three machine learning classifiers - K-Nearest Neighbours (K-NN), Random Forest, and Deep Neural Networks (DNN). The performance of each model was assessed using various metrics, including accuracy, ROC curve, confusion matrix, and precision, recall, and F1 score. Additional considerations such as data imbalance, dimensionality reduction, and model overfitting were also addressed.

The K-NN classifier achieved a maximum test accuracy of approximately 86.26% with one neighbour, and a ROC AUC of approximately 0.9 for all height categories. Dimensionality reduction using PCA was explored, but it decreased the accuracy of the K-NN classifier, suggesting that the curse of dimensionality was not a significant problem for this dataset. K-Fold cross-validation confirmed that the K-NN classifier with one neighbour did not overfit the data, as the variance in accuracy across folds was low.

The Random Forest classifier achieved a maximum test accuracy of approximately 86.75% with 100 trees and unlimited depth, and a ROC AUC of approximately 0.96 for all height categories - both metrics score higher than the K-classifier. Hyperparameter tuning was explored by varying the number of trees and maximum depth, with the best performanced observed at 100 trees and unlimited depth. K-Fold cross-validation confirmed that the Random Forest classifier did not overfit the data, as the variance in accuracy across folds was low. PCA was also applied to the data, which significantly reduced the computational time of the Random Forest classifier, but did not significantly decrease the accuracy.

A two hidden-layer DNN outperformed both the K-NN and Random Forest classifiers, achieving a maximum test accuracy of approximately 88.8% with a ROC AUC of approximately 0.98 for all height categories. The DNN also demonstrated low variance in accuracy across folds during K-Fold cross-validation, indicating that it did not overfit the data. A hyperparameter search was conducted to find the optimal hidden layer sizes, with the best performance observed at 192-192 hidden layers.

While all three classifiers demonstrated strong performance at classifying the height of table tennis players based on their swing data, the DNN emerged as the most effective model in terms of predictive power, and balance across metrics. Nonetheless, trade-offs in training time and interpretability should be considered when these three models in practice. Here is a list of the key findings:

- K-NN classifier achieved a maximum test accuracy of 86.26% with one neighbour, and a ROC AUC of 0.9.
- Random Forest classifier achieved a maximum test accuracy of 86.75% with 100 trees and unlimited depth, and a ROC AUC of 0.96.
- DNN classifier achieved a maximum test accuracy of 88.8% with two hidden layers (192-192), and a ROC AUC of 0.98.
- Dimensionality reduction using PCA did not significantly improve the performance of the K-NN or Random Forest classifiers, but reduced computational time.

- K-Fold cross-validation confirmed that all three classifiers did not overfit the data, as the variance in accuracy across folds was low.

Future improvement could involve experimenting with different DNN architectures and incorporations of different structures such as convolutional layers, which could further enhance the model's performance. More extensive hyperparameter tuning for the Random Forest classifier, such as exploring different criteria for splitting nodes and specifying the minimum samples per leaf, should be explored. The K-NN classifier could be improved by exploring different distance metrics, such as Manhattan or Minkowski distance.

9 Appendix

9.1 All Project Code

Listing 1: All project code

```
    #!/usr/bin/env python
# coding: utf-8

# ### Imports
#

# In[22]:


import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import LabelEncoder
import time
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score


# ### Data Exploration

# In[3]:


# Reload the CSV file
```

```
file_path = "assignTTSWING.csv"
df = pd.read_csv(file_path)


# In[4]:


# Filter only feature columns (exclude metadata and IDs)
feature_columns = [col for col in df.columns if col.startswith(('a_', 'g_', 'ax_',
    'ay_', 'az_', 'gx_', 'gy_', 'gz_'))]
features = df[feature_columns]

# Assign the labels
testmode_labels = df['testmode'] if 'testmode' in df.columns else None
gender_labels = df['gender'] if 'gender' in df.columns else None
age_labels = df['age'] if 'age' in df.columns else None
playYears_labels = df['playYears'] if 'playYears' in df.columns else None
height_labels = df['height'] if 'height' in df.columns else None
weight_labels = df['weight'] if 'weight' in df.columns else None
handedness_labels = df['handedness'] if 'handedness' in df.columns else None
holdRacketHanded_labels = df['holdRacketHanded'] if 'holdRacketHanded' in df.
    columns else None

# Combine all labels into a single DataFrame
labels_df = pd.DataFrame({
    'testmode': testmode_labels,
    'gender' : gender_labels,
    'age' : age_labels,
    'playYears' : playYears_labels,
    'height' : height_labels,
    'weight' : weight_labels,
    'handedness' : handedness_labels,
    'holdRacketHanded' : holdRacketHanded_labels
})


# In[5]:


# Convert all labels to string type
labels_df = labels_df.astype(str)


# In[6]:


# Count how many of each label we have
label_counts = labels_df.apply(pd.Series.value_counts).fillna(0).astype(int)

# Print the label counts
print("Label counts:")
print(label_counts)


# In[7]:

```

```
92
93  # Reveal the rows that are ??? for age, playYears, height, weight
94  missing_labels = labels_df[labels_df.isin(['???']).any(axis=1)]
95
96  # Print the rows with missing labels
97  print("\nRows with missing labels:")
98  print(missing_labels)
99
100 # Create a list of all the rows that are ??? for age, playYears, height, weight
101 missing_rows = missing_labels.index.tolist()
102
103 # Delete the rows with missing labels from the labels and features DataFrames
104 labels_df_cleaned = labels_df.drop(missing_rows)
105 features_cleaned = features.drop(missing_rows)
106
107
108 # In[8]:
109
110
111 # Recheck that the invalid rows are gone
112 # Count how many of each label we have
113 label_counts = labels_df_cleaned.apply(pd.Series.value_counts)
114
115 # Print the label counts
116 print("Label counts:")
117 print(label_counts)
118
119
120 # In[9]:
121
122
123 # Finally assign the X_train, X_test, X_val, y_train, y_test, y_val
124 # We will use 60% of the data for training, 20% for testing, and 20% for validation
125 # The y_train will be the height inside of the labels_df
126 X_train, X_temp, y_train, y_temp = train_test_split(features_cleaned,
        labels_df_cleaned['height'], test_size=0.4, random_state=1)
127 X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
        random_state=1)
128
129
130 # In[10]:
131
132
133 # Check the sizes of the splits
134 print(f"X_train size: {X_train.shape}")
135 print(f"X_val size: {X_val.shape}")
136 print(f"X_test size: {X_test.shape}")
137
138 # Check the sizes of the splits
139 print(f"y_train size: {y_train.shape}")
140 print(f"y_val size: {y_val.shape}")
141 print(f"y_test size: {y_test.shape}")
142
143 # Check the ratios of the splits
144 print(f"X_train ratio: {X_train.shape[0] / features_cleaned.shape[0]}")
145 print(f"X_val ratio: {X_val.shape[0] / features_cleaned.shape[0]}")
```

```
print(f"X_test ratio: {X_test.shape[0] / features_cleaned.shape[0]}")

# Check the ratios of the splits
print(f"y_train ratio: {y_train.shape[0] / labels_df_cleaned.shape[0]}")
print(f"y_val ratio: {y_val.shape[0] / labels_df_cleaned.shape[0]}")
print(f"y_test ratio: {y_test.shape[0] / labels_df_cleaned.shape[0]}")


# #### SMOTE

# In[11]:


# Check how many 'low', 'medium', and 'high' labels we have in the y_train
y_train_counts = y_train.value_counts()

# Print the y_train counts with percentage
y_train_counts_percentage = y_train_counts / y_train_counts.sum() * 100
print("\nY_train counts with percentage:")
print(y_train_counts_percentage)

print("\nY_train counts:")
print(y_train_counts)


# In[12]:


# Apply SMOTE only to training data
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)

# Keep validation and test sets unchanged
# X_val, y_val remain as they are
# X_test, y_test remain as they are

# Print class distributions
print("Original training set distribution:", pd.Series(y_train).value_counts())
print("Balanced training set distribution:", pd.Series(y_train_balanced).
    value_counts())
print("Validation set distribution:", pd.Series(y_val).value_counts())
print("Test set distribution:", pd.Series(y_test).value_counts())


# #### STANDARDIZEA THE FEATURES

# In[13]:


# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_balanced)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)


```

```
# In[ ]:


import matplotlib.pyplot as plt

# Plot histograms for all features before standardization
features_cleaned.hist(bins=30, figsize=(18, 12), layout=(int(np.ceil(len(
    features_cleaned.columns)/6)), 6))
plt.suptitle('Feature Distributions Before Standardization', fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.97])
plt.show()


# In[ ]:


import seaborn as sns
import matplotlib.pyplot as plt

corr = features_cleaned.corr()
plt.figure(figsize=(12, 10))
sns.heatmap(corr, cmap='coolwarm', center=0)
plt.title('Feature Correlation Matrix')
plt.show()


# ### PCA

# In[14]:


# Compute the PCA via the covariance matrix method

# Standardize the features
mean = features.mean()
std = features.std()
features_standardized = (features - mean) / std

# Compute the covariance matrix
cov_matrix = features_standardized.cov()

# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Sort the eigenvalues and eigenvectors
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

# Perform dimensionality reduction by multiplying the data by the top two
    eigenvectors
pca_features = np.dot(features_standardized, sorted_eigenvectors[:, :2])
pca_features_df = pd.DataFrame(pca_features, columns=['PC1', 'PC2'])

# Plot the PCA results
plt.figure(figsize=(10, 6))
```

```
sns.scatterplot(data=pca_features_df, x='PC1', y='PC2', hue=testmode_labels,
    palette='Set1', alpha=0.7)
plt.title('PCA of Features')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Test Mode')
plt.grid()
plt.savefig('pca_features.png')
plt.show()



# In[15]:


# Again but with feature changed into ndarray
features_np = features.values

# Standardize the features
mean = np.mean(features_np, axis=0)
std_dev = np.std(features_np, axis=0)
features_standardized_np = (features_np - mean) / std_dev

# Compute the covariance matrix
cov_matrix_np = np.cov(features_standardized_np.T)

# Compute the eigenvalues and eigenvectors
eigenvalues_np, eigenvectors_np = np.linalg.eig(cov_matrix_np)
sorted_indices = np.argsort(eigenvalues_np)[::-1]
eigenvalues = eigenvalues_np[sorted_indices]
eigenvectors = eigenvectors_np[:, sorted_indices]

# Perform dimensionality reduction by multiplying the data by the top two
    eigenvectors
reduced_features = np.dot(features_standardized_np, eigenvectors[:, :2])



# In[16]:


# Make a Scree graph for all eigenvalues
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(eigenvalues) + 1), eigenvalues, marker='o', linestyle='--')
plt.title('Scree Plot of Eigenvalues')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue')
plt.xticks(range(1, len(eigenvalues) + 1))
plt.grid()
plt.savefig('scree_plot.png')
plt.show()

# Make another Scree graph for all eigenvalues but the y-axis is the percentage of
    variance explained
plt.figure(figsize=(10, 6))
explained_variance = eigenvalues / np.sum(eigenvalues) * 100
```

```
plt.plot(range(1, len(explained_variance) + 1), explained_variance, marker='o', linestyle='--')
plt.title('Scree Plot of Explained Variance')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance (%)')
plt.xticks(range(1, len(explained_variance) + 1))
plt.grid()
plt.show()

# Make a cumulative explained variance plot
plt.figure(figsize=(10, 6))
cumulative_explained_variance = np.cumsum(explained_variance)
plt.plot(range(1, len(cumulative_explained_variance) + 1), cumulative_explained_variance, marker='o', linestyle='--')
plt.title('Cumulative Explained Variance')
plt.xlabel('Principal Component')
plt.ylabel('Cumulative Explained Variance (%)')
plt.xticks(range(1, len(cumulative_explained_variance) + 1))
plt.grid()
plt.show()

# Check that the percentages add up to 100%
total_variance = np.sum(explained_variance)
print(f"Total variance explained by all components: {total_variance:.2f}%")


# In[17]:


# Reduce the features to 16 dimensions using PCA
reduced_features_16 = np.dot(features_standardized_np, eigenvectors[:, :16])

# Plot the correlation matrix of the reduced 16D features
plt.figure(figsize=(12, 10))
sns.heatmap(np.corrcoef(reduced_features_16.T), annot=True, fmt=".2f", cmap='coolwarm', square=True, cbar_kws={"shrink": .8})
plt.title('Correlation Matrix of Reduced 16D Features')
plt.show()


# In[18]:


# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(features)

# Apply PCA to reduce dimensions to 2D for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)


# ### K-NN ALGORITHM

# #### K-NN WITH ORIGINAL DATASET

```

```
# In[19]:


def run_knn_from_split(X_train, X_test, y_train, y_test, n_neighbors=5, class_names=None, plot_confusion=False, verbose=False, label_name=""):
    # Fit KNN
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)

    # Accuracy and loss
    accuracy = accuracy_score(y_test, y_pred)
    loss = 1 - accuracy

    if verbose:
        print(f"\nK-NN Results for label: {label_name}")
        print(classification_report(y_test, y_pred, target_names=class_names))
        print(f"Accuracy: {accuracy:.4f}")
        print(f"Loss: {loss:.4f}")

    if plot_confusion:
        from sklearn.metrics import confusion_matrix
        cm = confusion_matrix(y_test, y_pred)
        plt.figure(figsize=(6, 5))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=class_names, yticklabels=class_names)
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.title(f'Confusion Matrix: {label_name}')
        plt.tight_layout()
        plt.show()

    return accuracy, loss, knn

# # Example usage:
# from sklearn.model_selection import train_test_split
# from sklearn.preprocessing import LabelEncoder

# label_of_interest = 'height'
# y = labels_df[label_of_interest].values
# le = LabelEncoder()
# y_encoded = le.fit_transform(y)
# class_names = le.classes_

# # Split once
# X_train, X_test, y_train, y_test = train_test_split(
#     X_scaled, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded
# )

# # Run KNN for k=1
# accuracy, loss, _ = run_knn_from_split(X_train, X_test, y_train, y_test, n_neighbors=1, class_names=class_names, label_name=label_of_interest)


# In[20]:

```

```

n_neighbors_list = [2**i for i in range(13)]
results = []

label_of_interest = 'height'  # any label you want

# Encode both training and test labels
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train_balanced)  # Fit on training data
y_test_encoded = le.transform(y_test)  # Transform test data using same encoder
class_names = le.classes_

for n in n_neighbors_list:
    try:
        accuracy, loss, knn = run_knn_from_split(
            X_train_scaled,
            X_test_scaled,  # Use scaled test data
            y_train_encoded,  # Use encoded training labels
            y_test_encoded,  # Use encoded test labels
            n_neighbors=n,
            class_names=class_names,
            label_name=label_of_interest
        )
        results.append({'n_neighbors': n, 'accuracy': accuracy, 'loss': loss, '
            knn_model': knn})
        print(f"n_neighbors={n}: accuracy={accuracy:.4f}, loss={loss:.4f}")
    except ValueError as e:
        print(f"n_neighbors={n}: Error - {e}")

# Plot accuracy vs n_neighbors
plt.figure(figsize=(8, 5))
plt.plot([r['n_neighbors'] for r in results], [r['accuracy'] for r in results],
    marker='o')
plt.xscale('log', base=2)
plt.xlabel('n_neighbors')
plt.ylabel('Accuracy')
plt.title(f'K-NN Accuracy vs n_neighbors ({label_of_interest})')
plt.grid(True)
plt.show()


# In[28]:


# Plot accuracy against the test AND training data
n_neighbors_list = [2**i for i in range(13)]
results = []

label_of_interest = 'height'  # any label you want

# Encode both training and test labels
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train_balanced)  # Fit on training data
y_test_encoded = le.transform(y_test)  # Transform test data using same encoder
class_names = le.classes_

```

```
for n in n_neighbors_list:
    try:
        knn = KNeighborsClassifier(n_neighbors=n)
        knn.fit(X_train_scaled, y_train_encoded)
        y_train_pred = knn.predict(X_train_scaled)
        y_test_pred = knn.predict(X_test_scaled)
        train_acc = accuracy_score(y_train_encoded, y_train_pred)
        test_acc = accuracy_score(y_test_encoded, y_test_pred)
        results.append({'n_neighbors': n, 'train_accuracy': train_acc, '
            test_accuracy': test_acc, 'knn_model': knn})
        print(f"n_neighbors={n}: train_acc={train_acc:.4f}, test_acc={test_acc:.4f}
            ")
    except ValueError as e:
        print(f"n_neighbors={n}: Error - {e}")

# Plot training and test accuracy vs n_neighbors
plt.figure(figsize=(8, 5))
plt.plot([r['n_neighbors'] for r in results], [r['train_accuracy'] for r in results
    ], marker='o', label='Train Accuracy')
plt.plot([r['n_neighbors'] for r in results], [r['test_accuracy'] for r in results
    ], marker='s', label='Test Accuracy')
plt.xscale('log', base=2)
plt.xlabel('n_neighbors')
plt.ylabel('Accuracy')
plt.title(f'K-NN Train/Test Accuracy vs n_neighbors ({label_of_interest})')
plt.legend()
plt.grid(True)
plt.show()


# ##### K-NN CROSS-VALIDATION
#
#

# In[22]:


from sklearn.model_selection import StratifiedKFold, cross_val_score
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.neighbors import KNeighborsClassifier

# Combine X_train and X_test for X input into K-Fold
X_kfold = np.vstack((X_train, X_test))
y_fold = np.hstack((y_train, y_test))

# Scale X_kfold
X_kfold_scaled = scaler.fit_transform(X_kfold)

k_values = [2**i for i in range(6)]
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# Store the results
results_kfold = []
all_scores = []  # Store all fold scores for each k

```

```
for k in k_values:
    pipeline = Pipeline([
        ('smote', SMOTE(random_state=42)),
        ('knn', KNeighborsClassifier(n_neighbors=k))
    ])
    scores = cross_val_score(pipeline, X_kfold_scaled, y_fold, cv=cv, scoring='
       accuracy')
    print(f"k={k}: mean accuracy={scores.mean():.4f} (+/- {scores.std():.4f})")
    results_kfold.append({
        'k': k,
        'mean_accuracy': scores.mean(),
        'std_accuracy': scores.std(),
        'min_accuracy': scores.min(),
        'max_accuracy': scores.max()
    })
    all_scores.append(scores)  # Save the array of scores for boxplot


# In[25]:


plt.figure(figsize=(8, 5))
plt.boxplot(
    all_scores,
    positions=range(len(k_values)),
    widths=0.5,
    patch_artist=True,
    boxprops=dict(facecolor='lightblue', color='blue'),
    whiskerprops=dict(color='black'),
    capprops=dict(color='black'),
    medianprops=dict(color='red')
)

plt.xticks(
    range(len(k_values)),
    [f"$2^{i}$" for i in range(len(k_values))]
)
plt.xlabel('k (number of neighbors)')
plt.ylabel('Accuracy (CV folds)')
plt.title('K-NN Accuracy Distribution with K-Fold Cross-Validation')
plt.grid(True, axis='y')
plt.show()


# #### K-NN WITH PCA REDUCTION

# In[27]:


# Compare K-NN performance with different PCA dimensions
pca_dimensions = [16, 14, 12, 10, 8, 6, 4, 2]
n_neighbors_list = [2**i for i in range(13)]
results_multi_pca = {d: [] for d in pca_dimensions}
computation_times = {}

# First run KNN on original data
```

```
start_time = time.time()
results_original = []

for n in n_neighbors_list:
    try:
        accuracy, loss, knn = run_knn_from_split(
            X_train_scaled,
            X_test_scaled,
            y_train_encoded,
            y_test_encoded,
            n_neighbors=n,
            class_names=class_names,
            label_name="Original (33D)"
        )
        results_original.append({'n_neighbors': n, 'accuracy': accuracy, 'loss':
            loss})
        print(f"Original data - n_neighbors={n}: accuracy={accuracy:.4f}")
    except ValueError as e:
        print(f"Original data - n_neighbors={n}: Error - {e}")

computation_times[33] = time.time() - start_time

# Prepare all PCA transformations first
pca_data = {}
for n_components in pca_dimensions:
    pca = PCA(n_components=n_components)
    X_train_pca = pca.fit_transform(X_train_scaled)
    X_test_pca = pca.transform(X_test_scaled)
    pca_data[n_components] = (X_train_pca, X_test_pca)

# Run KNN for each PCA dimension
for n_components in pca_dimensions:
    X_train_pca, X_test_pca = pca_data[n_components]

    # Start timing only K-NN part
    start_time = time.time()

    for n in n_neighbors_list:
        try:
            accuracy, loss, knn = run_knn_from_split(
                X_train_pca,
                X_test_pca,
                y_train_encoded,
                y_test_encoded,
                n_neighbors=n,
                class_names=class_names,
                label_name=f"PCA-{n_components}D"
            )
            results_multi_pca[n_components].append(
                {'n_neighbors': n, 'accuracy': accuracy, 'loss': loss}
            )
            print(f"PCA-{n_components}D - n_neighbors={n}: accuracy={accuracy:.4f}"
                )
        except ValueError as e:
            print(f"PCA-{n_components}D - n_neighbors={n}: Error - {e}")

```

```
    # Record only K-NN computation time
    computation_times[n_components] = time.time() - start_time

    # Plot results for this PCA dimension
    plt.plot([r['n_neighbors'] for r in results_multi_pca[n_components]],
             [r['accuracy'] for r in results_multi_pca[n_components]],
             marker='o', label=f'PCA ({n_components}D)', linestyle='--')

plt.xscale('log', base=2)
plt.xlabel('k (number of neighbors)')
plt.ylabel('Accuracy')
plt.title(f'Comparison of Different PCA Dimensions')
plt.grid(True)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

# Plot computation times
plt.figure(figsize=(10, 6))
sorted_dimensions = sorted(computation_times.keys(), reverse=True)  # Sort
    dimensions in descending order
times = [computation_times[d] for d in sorted_dimensions]
plt.plot(sorted_dimensions, times, marker='o')
plt.xlabel('Number of Dimensions')
plt.ylabel('Computation Time (seconds)')
plt.title('Computation Time vs Dimensionality')
plt.grid(True)
plt.show()

# Print computation times
print("\nComputation times:")
print(f"Original (33D): {computation_times[33]:.2f} seconds")
for dim in pca_dimensions:
    print(f"PCA-{dim}D: {computation_times[dim]:.2f} seconds")


# #### K-NN ROC curve

# In[17]:


from sklearn.model_selection import StratifiedKFold
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import LabelBinarizer
import numpy as np
import matplotlib.pyplot as plt

# Prepare data for cross-validation (already in your code)
X_kfold = np.vstack((X_train, X_test))
y_fold = np.hstack((y_train, y_test))
X_kfold_scaled = scaler.fit_transform(X_kfold)
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

```

```
# Store models and scores for k=1
models = []
scores = []

for train_idx, test_idx in cv.split(X_kfold_scaled, y_fold):
    X_tr, X_te = X_kfold_scaled[train_idx], X_kfold_scaled[test_idx]
    y_tr, y_te = y_fold[train_idx], y_fold[test_idx]
    # Pipeline with SMOTE and 1-NN
    pipeline = Pipeline([
        ('smote', SMOTE(random_state=42)),
        ('knn', KNeighborsClassifier(n_neighbors=1))
    ])
    pipeline.fit(X_tr, y_tr)
    acc = pipeline.score(X_te, y_te)
    models.append(pipeline)
    scores.append(acc)

# Find the best model (highest accuracy on its fold)
best_idx = np.argmax(scores)
best_model = models[best_idx]
print(f"Best fold accuracy: {scores[best_idx]:.4f}")

# Use the best model to predict probabilities on the validation set
y_val_proba = best_model.predict_proba(X_val_scaled)

# Binarize y_val for multiclass ROC
lb = LabelBinarizer()
y_val_binarized = lb.fit_transform(y_val)
if y_val_binarized.shape[1] == 1:
    y_val_binarized = np.hstack([1 - y_val_binarized, y_val_binarized])

# Plot ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(y_val_binarized.shape[1]):
    fpr, tpr, _ = roc_curve(y_val_binarized[:, i], y_val_proba[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw=2, label=f'Class {lb.classes_[i]} (AUC = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Best 1-NN Model (Validation Set)')
plt.legend(loc="lower right")
plt.grid(True)
plt.tight_layout()
plt.show()


# #### K-NN Confusion Matrix

# In[18]:


from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
741
742 # Predict on the validation set
743 y_val_pred = best_model.predict(X_val_scaled)
744
745 # If you used label encoding, ensure y_val_pred and y_val are comparable
746 # If y_val is not encoded, encode it with the same encoder as used in training
747 # Example:
748 # y_val_encoded = le.transform(y_val)
749
750 # Compute confusion matrix
751 cm = confusion_matrix(y_val, y_val_pred, labels=best_model.classes_)
752
753 # Plot confusion matrix
754 disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_model.
        classes_)
755 plt.figure(figsize=(6, 5))
756 disp.plot(cmap='Blues', values_format='d')
757 plt.title('Confusion Matrix: Best 1-NN Model (Validation Set)')
758 plt.tight_layout()
759 plt.show()
760
761
762 # In[19]:
763
764
765 # Normalized confusion matrix
766 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
767 import numpy as np
768
769 # Predict on the validation set
770 y_val_pred = best_model.predict(X_val_scaled)
771
772 # Compute confusion matrices (raw counts and normalized)
773 cm = confusion_matrix(y_val, y_val_pred, labels=best_model.classes_)
774 cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
775
776 # Create side-by-side plots
777 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
778
779 # Plot raw counts
780 disp1 = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_model.
        classes_)
781 disp1.plot(ax=ax1, cmap='Blues', values_format='d')
782 ax1.set_title('Confusion Matrix\n(Raw Counts)')
783
784 # Plot normalized values
785 disp2 = ConfusionMatrixDisplay(confusion_matrix=cm_normalized, display_labels=
        best_model.classes_)
786 disp2.plot(ax=ax2, cmap='Blues', values_format='.2%')
787 ax2.set_title('Confusion Matrix\n(Normalized by Row)')
788
789 plt.tight_layout()
790 plt.show()
791
792 # Print classification rates for each class
793 print("\nClassification rates per class:")
```

```
for i, class_name in enumerate(best_model.classes_):
    print(f"{class_name}: {cm_normalized[i][i]:.2%} correct")


# In[20]:


from sklearn.metrics import classification_report

# Predict on the validation set
y_val_pred = best_model.predict(X_val_scaled)

# Print precision, recall, and F1 score for each class
print("Precision, Recall, and F1 Score (Validation Set):\n")
print(classification_report(y_val, y_val_pred, target_names=best_model.classes_))


# ### RANDOM FOREST CLASSIFIER

# #### Random Forest on original dataset

# In[23]:


import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

# Define parameter ranges
n_estimators_list = [10, 50, 100, 200, 500]
max_depth_list = [None, 5, 10, 20, 30, 40]

# Store results
results = []

for n_estimators in n_estimators_list:
    for max_depth in max_depth_list:
        rf = RandomForestClassifier(
            n_estimators=n_estimators,
            max_depth=max_depth,
            criterion='gini',
            random_state=42
        )
        rf.fit(X_train_scaled, y_train_balanced)
        y_pred = rf.predict(X_test_scaled)
        acc = accuracy_score(y_test, y_pred)
        results.append({
            'n_estimators': n_estimators,
            'max_depth': max_depth if max_depth is not None else 'None',
            'accuracy': acc
        })
        print(f"n_estimators={n_estimators}, max_depth={max_depth}: accuracy={acc
            :.4f}")
```

```
849
850 # Convert results to DataFrame for heatmap
851 df_results = pd.DataFrame(results)
852 pivot_table = df_results.pivot(index='max_depth', columns='n_estimators', values='
        accuracy')
853
854 # Plot heatmap
855 plt.figure(figsize=(8, 5))
856 sns.heatmap(pivot_table, annot=True, fmt=".3f", cmap="YlGnBu")
857 plt.title("Random Forest Accuracy\n(varying n_estimators and max_depth)")
858 plt.xlabel("n_estimators")
859 plt.ylabel("max_depth")
860 plt.show()
861
862
863 # In[24]:
864
865
866 # Continue with more results
867
868 # Define parameter ranges
869 n_estimators_list = [800, 1400]
870 max_depth_list = [None, 5, 10, 20, 30, 40]
871
872 # Store results
873 results_2 = []
874
875 for n_estimators in n_estimators_list:
876     for max_depth in max_depth_list:
877         rf = RandomForestClassifier(
878             n_estimators=n_estimators,
879             max_depth=max_depth,
880             criterion='gini',
881             random_state=42
882         )
883         rf.fit(X_train_scaled, y_train_balanced)
884         y_pred = rf.predict(X_test_scaled)
885         acc = accuracy_score(y_test, y_pred)
886         results_2.append({
887             'n_estimators': n_estimators,
888             'max_depth': max_depth if max_depth is not None else 'None',
889             'accuracy': acc
890         })
891         print(f"n_estimators={n_estimators}, max_depth={max_depth}: accuracy={acc
            :.4f}")
892
893 # Combine results with results_2 to make a combined results
894 combined_results = results + results_2
895
896 # Convert results to DataFrame for heatmap
897 df_results = pd.DataFrame(combined_results)
898 pivot_table = df_results.pivot(index='max_depth', columns='n_estimators', values='
        accuracy')
899
900 # Plot heatmap
901 plt.figure(figsize=(8, 5))
```

```
sns.heatmap(pivot_table, annot=True, fmt=".3f", cmap="YlGnBu")
plt.title("Random Forest Accuracy\n(varying n_estimators and max_depth)")
plt.xlabel("n_estimators")
plt.ylabel("max_depth")
plt.show()


# In[33]:


# SAME AS ABOVE BUT WITH TRAINING AND TEST ACCURACY

# Define parameter ranges
n_estimators_list = [10, 50, 100, 200, 500, 800, 1400]
max_depth_list = [None, 5, 10, 20, 30, 40]

# Store results
results = []

for n_estimators in n_estimators_list:
    for max_depth in max_depth_list:
        rf = RandomForestClassifier(
            n_estimators=n_estimators,
            max_depth=max_depth,
            criterion='gini',
            random_state=42
        )
        rf.fit(X_train_scaled, y_train_balanced)
        y_train_pred = rf.predict(X_train_scaled)
        y_test_pred = rf.predict(X_test_scaled)
        train_acc = accuracy_score(y_train_balanced, y_train_pred)
        test_acc = accuracy_score(y_test, y_test_pred)
        results.append({
            'n_estimators': n_estimators,
            'max_depth': max_depth if max_depth is not None else 'None',
            'train_accuracy': train_acc,
            'test_accuracy': test_acc
        })
        print(f"n_estimators={n_estimators}, max_depth={max_depth}: train_acc={
            train_acc:.4f}, test_acc={test_acc:.4f}")



# In[34]:


# Convert results to DataFrame for heatmap
df_results = pd.DataFrame(results)

# Pivot for heatmaps
pivot_train = df_results.pivot(index='max_depth', columns='n_estimators', values='
    train_accuracy')
pivot_test = df_results.pivot(index='max_depth', columns='n_estimators', values='
    test_accuracy')

# Plot heatmaps
```

```
plt.figure(figsize=(16, 5))
plt.subplot(1, 2, 1)
sns.heatmap(pivot_train, annot=True, fmt=".3f", cmap="YlGnBu")
plt.title("Random Forest Train Accuracy")
plt.xlabel("n_estimators")
plt.ylabel("max_depth")

plt.subplot(1, 2, 2)
sns.heatmap(pivot_test, annot=True, fmt=".3f", cmap="YlGnBu")
plt.title("Random Forest Test Accuracy")
plt.xlabel("n_estimators")
plt.ylabel("max_depth")

plt.tight_layout()
plt.show()


# #### Random Forest on PCA reduced dataset

# In[ ]:


# List of dimensions to test (including original)
dimensions = [X_train_scaled.shape[1], 16, 8, 4, 2]
times = []
accuracies = []

for dim in dimensions:
    if dim == X_train_scaled.shape[1]:
        # Use original data
        X_train_dim = X_train_scaled
        X_test_dim = X_test_scaled
    else:
        # PCA reduction
        pca = PCA(n_components=dim, random_state=42)
        X_train_dim = pca.fit_transform(X_train_scaled)
        X_test_dim = pca.transform(X_test_scaled)

    # Measure time
    start = time.time()
    rf = RandomForestClassifier(n_estimators=200, max_depth=30, criterion='gini',
       random_state=42)
    rf.fit(X_train_dim, y_train_balanced)
    y_pred = rf.predict(X_test_dim)
    elapsed = time.time() - start
    acc = accuracy_score(y_test, y_pred)

    times.append(elapsed)
    accuracies.append(acc)
    print(f"Dimensions: {dim}, Time: {elapsed:.2f}s, Accuracy: {acc:.4f}")

# # Plot time vs dimensions
# plt.figure(figsize=(8, 5))
# plt.plot(dimensions, times, marker='o')
# plt.xlabel('Number of Dimensions')
# plt.ylabel('Time Taken (seconds)')
```

```
# plt.title('Random Forest: Time vs Number of Dimensions')
# plt.grid(True)
# plt.show()

# # Plot accuracy vs dimensions
# plt.figure(figsize=(8, 5))
# plt.plot(dimensions, accuracies, marker='o')
# plt.xlabel('Number of Dimensions')
# plt.ylabel('Accuracy')
# plt.title('Random Forest: Accuracy vs Number of Dimensions')
# plt.grid(True)
# plt.show()


# In[27]:


plt.figure(figsize=(8, 5))
ax1 = plt.gca()
color1 = 'tab:blue'
color2 = 'tab:orange'

# Plot time taken (left y-axis)
ax1.plot(dimensions, times, marker='o', color=color1, label='Time Taken (s)')
ax1.set_xlabel('Number of Dimensions')
ax1.set_ylabel('Time Taken (seconds)', color=color1)
ax1.tick_params(axis='y', labelcolor=color1)

# Create a second y-axis for accuracy
ax2 = ax1.twinx()
ax2.plot(dimensions, accuracies, marker='s', color=color2, label='Accuracy')
ax2.set_ylabel('Accuracy', color=color2)
ax2.tick_params(axis='y', labelcolor=color2)

plt.title('Random Forest: Time and Accuracy vs Number of Dimensions')
ax1.grid(True)
plt.show()


# #### Random Forest with cross-validation

# In[35]:


from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold
import matplotlib.pyplot as plt
import numpy as np

# Combine train and test sets
X_kfold = np.vstack((X_train, X_test))
y_fold = np.hstack((y_train, y_test))

max_depth_list = [5, 10, 20, 30, 40, None]
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
all_scores = []
```

```
1066
1067 for max_depth in max_depth_list:
1068     rf = RandomForestClassifier(
1069         n_estimators=100,
1070         max_depth=max_depth,
1071         criterion='gini',
1072         random_state=42
1073     )
1074     scores = cross_val_score(rf, X_kfold, y_fold, cv=cv, scoring='accuracy')
1075     all_scores.append(scores)
1076     print(f"max_depth={max_depth}: mean={scores.mean():.4f}, std={scores.std():.4f}
        ")
1077
1078 # Plot box-and-whisker plots with mean
1079 plt.figure(figsize=(8, 5))
1080 plt.boxplot(
1081     all_scores,
1082     labels=[str(md) for md in max_depth_list],
1083     patch_artist=True,
1084     showmeans=True,
1085     meanprops=dict(marker='o', markerfacecolor='red', markeredgecolor='black')
1086 )
1087 plt.xlabel('max_depth')
1088 plt.ylabel('Accuracy (CV folds)')
1089 plt.title('Random Forest 10-Fold CV Accuracy by max_depth')
1090 plt.grid(True, axis='y')
1091 plt.show()
1092
1093
1094 # #### Random Forest ROC Curve
1095
1096 # In[37]:
1097
1098
1099 from sklearn.ensemble import RandomForestClassifier
1100 from sklearn.metrics import roc_curve, auc
1101 from sklearn.preprocessing import LabelBinarizer
1102 import matplotlib.pyplot as plt
1103 import numpy as np
1104
1105 # Train the best model
1106 rf = RandomForestClassifier(n_estimators=100, max_depth=None, random_state=42)
1107 rf.fit(X_train_scaled, y_train_balanced)
1108
1109 # Predict probabilities on validation set
1110 y_val_proba = rf.predict_proba(X_val_scaled)
1111
1112 # Binarize the validation labels for multiclass ROC
1113 lb = LabelBinarizer()
1114 y_val_binarized = lb.fit_transform(y_val)
1115 if y_val_binarized.shape[1] == 1:
1116     y_val_binarized = np.hstack([1 - y_val_binarized, y_val_binarized])
1117
1118 # Plot ROC curve for each class
1119 plt.figure(figsize=(8, 6))
1120 for i in range(y_val_binarized.shape[1]):
```

```
    fpr, tpr, _ = roc_curve(y_val_binarized[:, i], y_val_proba[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw=2, label=f'Class {lb.classes_[i]} (AUC = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Random Forest (Validation Set)')
plt.legend(loc="lower right")
plt.grid(True)
plt.tight_layout()
plt.show()


# #### Random Forest Confusion Matrix

# In[39]:


from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

# Predict on the validation set
y_val_pred = rf.predict(X_val_scaled)

# Compute confusion matrices (raw counts and normalized)
cm = confusion_matrix(y_val, y_val_pred, labels=rf.classes_)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Create side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot raw counts
disp1 = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf.classes_)
disp1.plot(ax=ax1, cmap='Blues', values_format='d')
ax1.set_title('Confusion Matrix\n(Raw Counts)')

# Plot normalized values
disp2 = ConfusionMatrixDisplay(confusion_matrix=cm_normalized, display_labels=rf.
    classes_)
disp2.plot(ax=ax2, cmap='Blues', values_format='.2%')
ax2.set_title('Confusion Matrix\n(Normalized by Row)')

plt.tight_layout()
plt.show()

# Print classification rates for each class
print("\nClassification rates per class:")
for i, class_name in enumerate(rf.classes_):
    print(f"{class_name}: {cm_normalized[i][i]:.2%} correct")


# #### Random Forest Precision, Recall, F1

```

```
# In[40]:


from sklearn.metrics import classification_report

# Predict on the validation set
y_val_pred = rf.predict(X_val_scaled)

# Print precision, recall, and F1 score for each class
print("Precision, Recall, and F1 Score (Validation Set):\n")
print(classification_report(y_val, y_val_pred, target_names=rf.classes_))


# ### Deep Neural Net

# #### Training and testing two hidden layers with Cross Entropy (CE) and L2
    regularisation

# In[ ]:


import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

# Define the model
class MLP(nn.Module):
    def __init__(self, input_dim, hidden1, hidden2, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, num_classes)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Set your variables
hidden1 = 128  # number of nodes in first hidden layer
hidden2 = 128  # number of nodes in second hidden layer
lambda_l2 = 0.00  # L2 regularization strength
epochs = 100
batch_size = 16
learning_rate = 0.001
epochs_per_round = 100 # number of epochs to train before asking for user input

# Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Encode labels
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train_balanced)
```

```
y_test_enc = le.transform(y_test)
num_classes = len(le.classes_)

# Convert data to torch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_enc, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_enc, dtype=torch.long)

# Dataset and DataLoader
train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
    shuffle=True)

model = MLP(X_train_scaled.shape[1], hidden1, hidden2, num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=lambda_l2
    )

# Training loop
total_epochs = 0
while True:
    for epoch in range(epochs_per_round):
        model.train()
        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            optimizer.zero_grad()
            outputs = model(xb)
            loss = criterion(outputs, yb)
            loss.backward()
            optimizer.step()
        total_epochs += 1
        if (epoch+1) % 5 == 0 or epoch == 0:
            print(f"Epoch {total_epochs}, Loss: {loss.item():.4f}")

    # Ask user if they want to continue
    cont = input("Continue training for another 30 epochs? (y/n): ").strip().lower
        ()
    if cont != 'y':
        break
    # Ask for new learning rate
    try:
        new_lr = float(input("Enter new learning rate for next 30 epochs (current:
            {:.5f}): ".format(optimizer.param_groups[0]['lr'])))
        for param_group in optimizer.param_groups:
            param_group['lr'] = new_lr
    except Exception:
        print("Invalid input. Keeping previous learning rate.")

# Evaluate on test set
model.eval()
with torch.no_grad():
    X_test_tensor = X_test_tensor.to(device)
    y_test_tensor = y_test_tensor.to(device)
```

```
    outputs = model(X_test_tensor)
    _, predicted = torch.max(outputs, 1)
    test_acc = accuracy_score(y_test_tensor.cpu(), predicted.cpu())
    print(f"Test accuracy: {test_acc:.4f}")


# #### Deep neural net 5-fold cross-validation

# In[ ]:


# --- Prepare data ---
# Combine train and test sets for K-Fold
X_kfold = np.vstack((X_train, X_test))
y_fold = np.hstack((y_train, y_test))
# Scale X_kfold
X_kfold_scaled = scaler.fit_transform(X_kfold)

X = X_kfold_scaled
le = LabelEncoder()
y = le.fit_transform(y_fold)
num_classes = len(le.classes_)

# --- Model definition ---
class MLP(nn.Module):
    def __init__(self, input_dim, hidden1, hidden2, num_classes):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, num_classes)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# --- Hyperparameters ---
hidden_sizes = [128, 160, 192]
lambda_l2 = 0.00
epochs = 100
batch_size = 16
learning_rate = 0.001
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# --- Cross-validation setup ---
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
results = {}  # {(h1, h2): [fold_accuracies]}

for h1 in hidden_sizes:
    for h2 in hidden_sizes:
        fold_accuracies = []
        print(f"\nTraining MLP with hidden sizes: ({h1}, {h2})")
        for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
            X_tr, X_val = X[train_idx], X[val_idx]
            y_tr, y_val = y[train_idx], y[val_idx]
```

```
            # Convert to tensors
            X_tr_tensor = torch.tensor(X_tr, dtype=torch.float32)
            y_tr_tensor = torch.tensor(y_tr, dtype=torch.long)
            X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
            y_val_tensor = torch.tensor(y_val, dtype=torch.long)

            train_dataset = torch.utils.data.TensorDataset(X_tr_tensor, y_tr_tensor
                )
            train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=
                batch_size, shuffle=True)

            model = MLP(X.shape[1], h1, h2, num_classes).to(device)
            criterion = nn.CrossEntropyLoss()
            optimizer = optim.Adam(model.parameters(), lr=learning_rate,
                weight_decay=lambda_l2)

            # Training loop
            for epoch in range(epochs):
                model.train()
                for xb, yb in train_loader:
                    xb, yb = xb.to(device), yb.to(device)
                    optimizer.zero_grad()
                    outputs = model(xb)
                    loss = criterion(outputs, yb)
                    loss.backward()
                    optimizer.step()
                if (epoch+1) % 10 == 0 or epoch == 0:
                    print(f"  Fold {fold+1}, Epoch {epoch+1}/{epochs}, Loss: {loss.
                        item():.4f}")

            # Validation
            model.eval()
            with torch.no_grad():
                X_val_tensor = X_val_tensor.to(device)
                y_val_tensor = y_val_tensor.to(device)
                outputs = model(X_val_tensor)
                _, predicted = torch.max(outputs, 1)
                acc = accuracy_score(y_val_tensor.cpu(), predicted.cpu())
                fold_accuracies.append(acc)
                print(f"  Fold {fold+1}: Accuracy = {acc:.4f}")

        results[(h1, h2)] = fold_accuracies



# In[66]:


# --- Box and whisker plot ---
plt.figure(figsize=(10, 6))
labels = []
data = []
for (h1, h2), accs in results.items():
    labels.append(f"{h1}-{h2}")
    data.append(accs)
```

```
plt.boxplot(data, labels=labels, patch_artist=True)
plt.xlabel("Hidden Layer Sizes (h1-h2)")
plt.ylabel("Test Accuracy")
plt.title("5-Fold CV Accuracy for Different MLP Hidden Layer Sizes")
plt.grid(True, axis='y')
plt.show()


# #### Training best model (192-192)

# In[ ]:


# Hyperparameters
input_dim = X_train_scaled.shape[1]
hidden1 = 192
hidden2 = 192
epochs = 500
learning_rate = 0.001
lambda_l2 = 0.00
patience = 1000
batch_size = 16
prompt_every = 20  # Number of epochs to train before prompting user
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Model definition
class MLP(nn.Module):
    def __init__(self, input_dim, hidden1, hidden2, num_classes):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, num_classes)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Encode labels
le = LabelEncoder()
y_train_enc = le.fit_transform(y_train_balanced)
y_val_enc = le.transform(y_val)
num_classes = len(le.classes_)

# Convert data to torch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_enc, dtype=torch.long)
X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val_enc, dtype=torch.long)

# DataLoader
train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
    shuffle=True)

```

```
model = MLP(input_dim, hidden1, hidden2, num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=lambda_l2)

# Early stopping variables
best_val_acc = 0.0
epochs_no_improve = 0
best_model_state = None

train_accuracies = []
val_accuracies = []

total_epochs = 0
while total_epochs < epochs:
    for _ in range(prompt_every):
        if total_epochs >= epochs:
            break
        model.train()
        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            optimizer.zero_grad()
            outputs = model(xb)
            loss = criterion(outputs, yb)
            loss.backward()
            optimizer.step()
        # Training accuracy
        model.eval()
        with torch.no_grad():
            train_outputs = model(X_train_tensor.to(device))
            _, train_pred = torch.max(train_outputs, 1)
            train_acc = accuracy_score(y_train_tensor.cpu(), train_pred.cpu())
            train_accuracies.append(train_acc)
            # Validation accuracy
            val_outputs = model(X_val_tensor.to(device))
            _, val_pred = torch.max(val_outputs, 1)
            val_acc = accuracy_score(y_val_tensor.cpu(), val_pred.cpu())
            val_accuracies.append(val_acc)
        print(f"Epoch {total_epochs+1}, Loss: {loss.item():.4f}, Train Acc: {train_acc:.4f}, Val Acc: {val_acc:.4f}, Learning Rate: {optimizer.param_groups[0]['lr']:.5f}")
        # Early stopping check
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            epochs_no_improve = 0
            best_model_state = model.state_dict()
        else:
            epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print(f"Early stopping at epoch {total_epochs+1}. Best Val Acc: {best_val_acc:.4f}")
            break
        total_epochs += 1
    # Prompt user for learning rate change or to continue
    if total_epochs < epochs and epochs_no_improve < patience:
```

```
        cont = input(f"Continue training for another {prompt_every} epochs? (y/n): ").strip().lower()
        if cont != 'y':
            break
        try:
            new_lr = float(input(f"Enter new learning rate (current: {optimizer.param_groups[0]['lr']:.5f}): "))
            for param_group in optimizer.param_groups:
                param_group['lr'] = new_lr
        except Exception:
            print("Invalid input. Keeping previous learning rate.")



# In[33]:


import re
import matplotlib.pyplot as plt

# Path to your log file
log_path = "dnn_train_test_results.yaml"

# Prepare lists
epochs = []
train_accs = []
val_accs = []
lrs = []

# Regex to extract values
epoch_re = re.compile(
    r"Epoch (\d+), Loss: [\d\.eE+-]+, Train Acc: ([\d\.eE+-]+), Val Acc: ([\d\.eE+-]+), Learning Rate: ([\d\.eE+-]+)"
)

with open(log_path, "r") as f:
    for line in f:
        match = epoch_re.search(line)
        if match:
            epoch = int(match.group(1))
            train_acc = float(match.group(2))
            val_acc = float(match.group(3))
            lr = float(match.group(4))
            epochs.append(epoch)
            train_accs.append(train_acc)
            val_accs.append(val_acc)
            lrs.append(lr)

# Plot
fig, ax1 = plt.subplots(figsize=(10, 6))

color1 = 'tab:blue'
color2 = 'tab:orange'
color3 = 'tab:green'

# Plot train/val accuracy
```

```
ax1.plot(epochs, train_accs, label='Train Accuracy', color=color1)
ax1.plot(epochs, val_accs, label='Validation Accuracy', color=color2)
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Accuracy')
ax1.legend(loc='upper left')
ax1.grid(True, axis='y')

# Plot learning rate on right y-axis (log10 scale)
ax2 = ax1.twinx()
ax2.plot(epochs, lrs, label='Learning Rate', color=color3, linestyle='--')
ax2.set_ylabel('Learning Rate (log10 scale)', color=color3)
ax2.tick_params(axis='y', labelcolor=color3)
ax2.set_yscale('log')

plt.title('Train/Validation Accuracy and Learning Rate vs Epoch')
plt.tight_layout()
plt.show()


# In[30]:


# Load best model
if best_model_state is not None:
    model.load_state_dict(best_model_state)


# #### ROC Curve

# In[34]:


from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import LabelBinarizer
import matplotlib.pyplot as plt
import numpy as np
import torch

# Ensure model is in eval mode and on correct device
model.eval()
X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32).to(device)

# Get predicted probabilities from the DNN
with torch.no_grad():
    logits = model(X_val_tensor)
    y_val_proba = torch.softmax(logits, dim=1).cpu().numpy()

# Binarize the validation labels for multiclass ROC
lb = LabelBinarizer()
y_val_binarized = lb.fit_transform(y_val)
if y_val_binarized.shape[1] == 1:
    y_val_binarized = np.hstack([1 - y_val_binarized, y_val_binarized])

# Plot ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(y_val_binarized.shape[1]):
```

```
    fpr, tpr, _ = roc_curve(y_val_binarized[:, i], y_val_proba[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw=2, label=f'Class {lb.classes_[i]} (AUC = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for DNN (Validation Set)')
plt.legend(loc="lower right")
plt.grid(True)
plt.tight_layout()
plt.show()


# #### Confusion Matrix

# In[36]:


from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

# Predict on the validation set using the best DNN model
if best_model_state is not None:
    model.load_state_dict(best_model_state)
model.eval()
with torch.no_grad():
    X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32).to(device)
    logits = model(X_val_tensor)
    y_val_pred = torch.argmax(logits, dim=1).cpu().numpy()

# Use the same label encoder as during training for class names
class_names = le.classes_

# Compute confusion matrices (raw counts and normalized)
cm = confusion_matrix(y_val_enc, y_val_pred, labels=range(len(class_names)))
cm_normalized = cm.astype('float') / cm.sum(axis=1, keepdims=True)

# Create side-by-side plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot raw counts
disp1 = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp1.plot(ax=ax1, cmap='Blues', values_format='d')
ax1.set_title('Confusion Matrix\n(Raw Counts)')

# Plot normalized values
disp2 = ConfusionMatrixDisplay(confusion_matrix=cm_normalized, display_labels=
    class_names)
disp2.plot(ax=ax2, cmap='Blues', values_format='.2%')
ax2.set_title('Confusion Matrix\n(Normalized by Row)')

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

```

# Print classification rates for each class
print("\nClassification rates per class:")
for i, class_name in enumerate(class_names):
    print(f"{class_name}: {cm_normalized[i][i]:.2%} correct")


# #### F1, Precision, and Recall

# In[37]:


from sklearn.metrics import classification_report

# Predict on the validation set using the best DNN model
if best_model_state is not None:
    model.load_state_dict(best_model_state)
model.eval()
with torch.no_grad():
    X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32).to(device)
    logits = model(X_val_tensor)
    y_val_pred = torch.argmax(logits, dim=1).cpu().numpy()

# Use the same label encoder as during training for class names
class_names = le.classes_

# Print precision, recall, and F1 score for each class
print("Precision, Recall, and F1 Score (Validation Set):\n")
print(classification_report(y_val_enc, y_val_pred, target_names=class_names))
```