# Classical Machine Learning - Classification of Digits Dataset

## Introduction

Machine learning models are widely used in classification tasks, and their performance depends on various factors, including feature selection, dimensionality reduction, and hyperparameter tuning.

This study applies **Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Decision Trees (DT), Random**

**Forests (RF), Logistic Regression (LR), and Naïve Bayes (NB)** to classify images in the **Digits dataset** from sklearn.datasets .

Prior studies have demonstrated the effectiveness of ensemble methods such as **Random Forest** in

classification tasks (Jovanovic et al., 2024), while dimensionality reduction techniques like **Principal Component Analysis (PCA)** have been found to improve computational efficiency but may also remove essential features (Zivkovic et al., 2024).

import numpy as np import pandas as pd import matplotlib.pyplot as plt import seaborn as sns from sklearn.model\_selection import train\_test\_split from sklearn.metrics import accuracy\_score, balanced\_accuracy\_score, roc\_auc\_score, confusion\_matrix, cla from sklearn.preprocessing import StandardScaler from sklearn.decomposition import PCA from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier from sklearn.tree import DecisionTreeClassifier from sklearn.ensemble import RandomForestClassifier from sklearn.linear\_model import LogisticRegression from sklearn.naive\_bayes import GaussianNB from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis # Fixed from sklearn.feature\_selection import SelectFromModel # Fixed import joblib import warnings warnings.filterwarnings('ignore')

**Loading the Dataset and Exploring it**

The dataset consists of **8x8 pixel grayscale images** representing handwritten digits (0-9). Each image is flattened into a 64-dimensional feature vector.

We split the dataset into **80% training and 20% testing**.

from sklearn.datasets import load\_digits import numpy as np

# Load dataset containing images of handwritten digits digits = load\_digits()

X, y = digits.data, digits.target

# X holds numerical data representing images, y contains the digit labels (0-9) print(f'Dataset Shape: {X.shape}') # Check dataset size

print(f'Unique Classes: {np.unique(y)}') # Confirm all digit classes are included

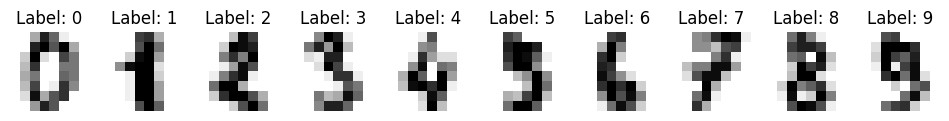
Dataset Shape: (1797, 64)

Unique Classes: [0 1 2 3 4 5 6 7 8 9]

# Visualising sample images from the dataset fig, axes = plt.subplots(1, 10, figsize=(12, 6))

for i, ax in enumerate(axes.flat):

ax.set\_axis\_off() ax.imshow(digits.images[i], cmap='gray\_r', interpolation='none') ax.set\_title(f'Label: {digits.target[i]}', fontsize=12) plt.show()



## Data Preprocessing

To prepare the dataset, we apply:

**Train-Test Split:** 80% training, 20% testing ( train\_test\_split ).

**Feature Scaling:** Standardization using StandardScaler .

**Dimensionality Reduction (if applicable):** PCA to assess its impact.

PCA has been shown to enhance computational efficiency but may lead to performance degradation if key features are lost (Ahlawat & Khurana, 2024).

# Splitting data into training (80%) and testing (20%) sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y) print(f'Training Set: {X\_train.shape}, Testing Set: {X\_test.shape}')

Training Set: (1437, 64), Testing Set: (360, 64)

# Classification

Training Models - Baseline Performance

## Machine Learning Models

The following classifiers are implemented:

**Support Vector Machines (SVM)** - Suitable for high-dimensional spaces and effective in capturing complex decision boundaries, which makes it well-suited for handwritten digit classification. In particular, **Support Vector Classification** (SVC) was employed in our approach, leveraging its ability to find an optimal hyperplane for distinguishing digit classes with strong generalisation performance [Mahardhika et al., 2025].

**K-Nearest Neighbors (KNN)** - KNN performs well in small datasets and is sensitive to the local data structure, making it a useful method when decision boundaries are non-linear [Jovanovic et al., 2024]. **Decision Trees (DT)** - Highly interpretable and fast to train, Decision Trees can handle both categorical and numerical data, though they are prone to overfitting, particularly with small datasets [Mustafa, 2024]. **Random Forest (RF)** - An ensemble of decision trees that is robust against overfitting and widely used in machine learning, especially when dealing with noisy data [Jovanovic et al., 2024].

**Logistic Regression (LR)** - Effective for binary classification, Logistic Regression can perform reasonably well in multi-class settings but may struggle with complex, high-dimensional data like images [Du et al., 2024].

**Naïve Bayes (NB)** - A simple, probabilistic classifier that assumes independent features, making it computationally efficient, but it can struggle with highly correlated features often found in images [Du et al., 2024].

Each model is trained **with and without PCA** to explore the effects of dimensionality reduction.

### 1. Support Vector Machine (SVM)

# Train SVM on the dataset (no dimensionality reduction) svm = SVC(probability=True) svm.fit(X\_train, y\_train)

# Save the trained model joblib.dump(svm, "svm\_no\_dr.pkl")

['svm\_no\_dr.pkl']

### 2. K-Nearest Neighbors (KNN) - No Dimensionality Reduction

# Train KNN on the original dataset (no dimensionality reduction) knn = KNeighborsClassifier(n\_neighbors=5) knn.fit(X\_train, y\_train)

# Save the trained model joblib.dump(knn, "knn\_no\_dr.pkl")

['knn\_no\_dr.pkl']

### 3. Decision Tree - No Dimensionality Reduction

# Train Decision Tree without dimensionality reduction dt = DecisionTreeClassifier(random\_state=42) dt.fit(X\_train, y\_train)

# Save the trained model joblib.dump(dt, "dt\_no\_dr.pkl")

['dt\_no\_dr.pkl']

### 4. Decision Tree - PCA

# Apply PCA to reduce dimensionality to 30 components pca = PCA(n\_components=30)

X\_train\_pca = pca.fit\_transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

# Train Decision Tree on PCA-transformed data dt\_pca = DecisionTreeClassifier(random\_state=42) dt\_pca.fit(X\_train\_pca, y\_train)

# Save PCA model and trained Decision Tree joblib.dump(pca, "pca\_dt.pkl") joblib.dump(dt\_pca, "dt\_pca.pkl")

['dt\_pca.pkl']

### 5. Random Forest (RF) - No Dimensionality Reduction

# Train Random Forest without dimensionality reduction rf = RandomForestClassifier(n\_estimators=100, random\_state=42) rf.fit(X\_train, y\_train)

# Save the trained model joblib.dump(rf, "rf\_no\_dr.pkl")

['rf\_no\_dr.pkl']

### 6. Random Forest (RF) - PCA

# Apply PCA (reduce to 30 components) pca\_rf = PCA(n\_components=30)

X\_train\_pca\_rf = pca\_rf.fit\_transform(X\_train)

X\_test\_pca\_rf = pca\_rf.transform(X\_test)

# Train Random Forest with PCA

rf\_pca = RandomForestClassifier(n\_estimators=100, random\_state=42) rf\_pca.fit(X\_train\_pca\_rf, y\_train)

# Save PCA model and trained Random Forest with PCA joblib.dump(pca\_rf, "pca\_rf.pkl") joblib.dump(rf\_pca, "rf\_pca.pkl")

['rf\_pca.pkl']

### 7. Logistic Regression - No dimensionality reduction

# Train Logistic Regression without dimensionality reduction

lr = LogisticRegression(max\_iter=1000) # Increased iterations for convergence lr.fit(X\_train, y\_train)

# Save the trained model joblib.dump(lr, "lr\_no\_dr.pkl")

['lr\_no\_dr.pkl']

### 8. Naïve Bayes - Without Dimensionality Reduction

# Train Naïve Bayes without dimensionality reduction nb = GaussianNB() nb.fit(X\_train, y\_train)

# Save the trained model joblib.dump(nb, "nb\_no\_dr.pkl")

['nb\_no\_dr.pkl']

### 9. Naïve Bayes - PCA

# Apply PCA to reduce dimensionality to 30 components pca = PCA(n\_components=30)

X\_train\_pca = pca.fit\_transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

# Train Naïve Bayes on PCA-transformed data nb\_pca = GaussianNB() nb\_pca.fit(X\_train\_pca, y\_train)

# Save PCA model and trained Naïve Bayes joblib.dump(pca, "pca\_nb.pkl") joblib.dump(nb\_pca, "nb\_pca.pkl")

['nb\_pca.pkl']

# Evaluation - Baseline Performance

## Model Evaluation

We assess each classifier using:

**Balanced Accuracy**

**ROC-AUC (macro-averaged)**

**Confusion Matrices**

Random Forest have shown strong performance in prior research (Edafetanure-Ibeh, 2024). SVM performance varies significantly based on kernel choice (Wang et al., 2024).

### 1. Support Vector Machine (SVM) - No Dimensionality Reduction

# Load trained SVM model (No Dimensionality Reduction) svm = joblib.load("svm\_no\_dr.pkl")

# Make predictions y\_pred\_svm = svm.predict(X\_test) y\_prob\_svm = svm.predict\_proba(X\_test)

# Evaluate performance results\_svm = {

"SVM (No Dimensionality Reduction)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_svm),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_svm),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_svm, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_svm)

}

}

# Print evaluation results for name, metrics in results\_svm.items():

print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}")

SVM (No Dimensionality Reduction):

Accuracy: 0.9917

Balanced Accuracy: 0.9915

ROC AUC: 0.9999

### 2. K-Nearest Neighbors (No Dimensionality Reduction)

# Load trained KNN model (No Dimensionality Reduction) knn = joblib.load("knn\_no\_dr.pkl")

# Make predictions y\_pred\_knn = knn.predict(X\_test) y\_prob\_knn = knn.predict\_proba(X\_test)

# Evaluate performance results\_knn = {

"KNN (No Dimensionality Reduction)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_knn),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_knn),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_knn, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_knn)

}

}

# Print evaluation results for name, metrics in results\_knn.items():

print(f"\n{name}:")

print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}")

KNN (No Dimensionality Reduction):

Accuracy: 0.9833

Balanced Accuracy: 0.9831

ROC AUC: 0.9983

### 3 and 4. Decision Tree with and without PCA

# Load trained models dt = joblib.load("dt\_no\_dr.pkl") dt\_pca = joblib.load("dt\_pca.pkl") pca = joblib.load("pca\_dt.pkl")

# Make predictions without PCA y\_pred\_dt = dt.predict(X\_test) y\_prob\_dt = dt.predict\_proba(X\_test) # Apply PCA to test data and make predictions with PCA X\_test\_pca = pca.transform(X\_test) y\_pred\_dt\_pca = dt\_pca.predict(X\_test\_pca) y\_prob\_dt\_pca = dt\_pca.predict\_proba(X\_test\_pca)

# Evaluate performance results\_dt = {

"Decision Tree (No PCA)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_dt),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_dt),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_dt, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_dt)

},

"Decision Tree (With PCA)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_dt\_pca),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_dt\_pca),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_dt\_pca, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_dt\_pca)

}

}

# Print evaluation results for name, metrics in results\_dt.items():

print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}")

Decision Tree (No PCA):

Accuracy: 0.8250

Balanced Accuracy: 0.8242

ROC AUC: 0.9024

Decision Tree (With PCA):

Accuracy: 0.8472

Balanced Accuracy: 0.8462

ROC AUC: 0.9146

### 5 and 6. Random Forest (RF) with and without PCA

# Load trained models rf = joblib.load("rf\_no\_dr.pkl") rf\_pca = joblib.load("rf\_pca.pkl") pca\_rf = joblib.load("pca\_rf.pkl")

# Make predictions without PCA y\_pred\_rf = rf.predict(X\_test) y\_prob\_rf = rf.predict\_proba(X\_test)

# Apply PCA to test data and make predictions with PCA X\_test\_pca\_rf = pca\_rf.transform(X\_test) y\_pred\_rf\_pca = rf\_pca.predict(X\_test\_pca\_rf) y\_prob\_rf\_pca = rf\_pca.predict\_proba(X\_test\_pca\_rf)

# Evaluate performance results\_rf = {

"Random Forest (No PCA)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_rf),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_rf),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_rf, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_rf)

},

"Random Forest (With PCA)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_rf\_pca),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_rf\_pca),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_rf\_pca, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_rf\_pca)

}

}

# Print evaluation results for name, metrics in results\_rf.items():

print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}")

Random Forest (No PCA):

Accuracy: 0.9611

Balanced Accuracy: 0.9607

ROC AUC: 0.9992

Random Forest (With PCA):

Accuracy: 0.9722

Balanced Accuracy: 0.9718

ROC AUC: 0.9992

### 7. Logistic Regression - No dimensionality reduction

# Make predictions with Logistic Regression y\_pred\_lr = lr.predict(X\_test) y\_prob\_lr = lr.predict\_proba(X\_test)

# Evaluate performance results\_lr = {

"Logistic Regression": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_lr),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_lr),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_lr, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_lr)

}

}

# Print evaluation results for name, metrics in results\_lr.items():

print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}")

Logistic Regression:

Accuracy: 0.9583

Balanced Accuracy: 0.9580

ROC AUC: 0.9989

### 8 and 9.Naïve Bayes - With and Without PCA

# Load trained models nb = joblib.load("nb\_no\_dr.pkl") nb\_pca = joblib.load("nb\_pca.pkl") pca = joblib.load("pca\_nb.pkl")

# Make predictions without PCA y\_pred\_nb = nb.predict(X\_test) y\_prob\_nb = nb.predict\_proba(X\_test)

# Apply PCA to test data and make predictions with PCA X\_test\_pca = pca.transform(X\_test) y\_pred\_nb\_pca = nb\_pca.predict(X\_test\_pca) y\_prob\_nb\_pca = nb\_pca.predict\_proba(X\_test\_pca)

# Evaluate performance results\_nb = {

"Naïve Bayes (No PCA)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_nb),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_nb),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_nb, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_nb)

},

"Naïve Bayes (With PCA)": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_nb\_pca),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_nb\_pca),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_nb\_pca, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_nb\_pca)

}

}

# Print evaluation results for name, metrics in results\_nb.items():

print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}")

print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}")

Naïve Bayes (No PCA):

Accuracy: 0.8111

Balanced Accuracy: 0.8103

ROC AUC: 0.9705

Naïve Bayes (With PCA):

Accuracy: 0.9556

Balanced Accuracy: 0.9555

ROC AUC: 0.9986

# Hyperparameter Tuning

To optimise model performance, we apply **GridSearchCV** for hyperparameter tuning. This allows us to systematically evaluate different hyperparameter configurations and identify the optimal settings.

Tuned models:

**SVM**: Kernel selection and C parameter tuning (Wang et al., 2024)

**KNN**: Number of neighbours and weight functions (Jovanovic et al., 2024)

**Random Forest**: Number of estimators and max depth tuning (Edafetanure-Ibeh, 2024)

Each model's performance is evaluated using **Balanced Accuracy and ROC-AUC** to determine improvements over the baseline.

from sklearn.model\_selection import GridSearchCV

## 1. Support Vector Machine (SVM) with GridSearch

# Define hyperparameter grid for SVM param\_grid\_svm = {

'C': [0.1, 1, 10, 100],

'kernel': ['linear', 'rbf', 'poly'],

'gamma': ['scale', 'auto']

}

# Perform grid search with cross-validation grid\_search\_svm = GridSearchCV( estimator=SVC(probability=True), param\_grid=param\_grid\_svm, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on training data grid\_search\_svm.fit(X\_train, y\_train)

# Retrieve the best model and parameters best\_svm = grid\_search\_svm.best\_estimator\_

print("Best parameters for SVM:", grid\_search\_svm.best\_params\_)

# Save the best SVM model

joblib.dump(best\_svm, "svm\_no\_dr\_tuned.pkl")

Best parameters for SVM: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}

['svm\_no\_dr\_tuned.pkl']

## 2. K-Nearest Neighbour (KNN) with GridSearch

# Define hyperparameter grid for KNN param\_grid\_knn = {

'n\_neighbors': [3, 5, 7, 9, 11],

'weights': ['uniform', 'distance'],

'metric': ['euclidean', 'manhattan']

}

# Perform grid search with cross-validation grid\_search\_knn = GridSearchCV( estimator=KNeighborsClassifier(), param\_grid=param\_grid\_knn, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on training data grid\_search\_knn.fit(X\_train, y\_train)

# Retrieve the best model and parameters best\_knn = grid\_search\_knn.best\_estimator\_ print("Best parameters for KNN:", grid\_search\_knn.best\_params\_)

# Save the best KNN model joblib.dump(best\_knn, "knn\_no\_dr\_tuned.pkl") Best parameters for KNN: {'metric': 'euclidean', 'n\_neighbors': 3, 'weights': 'uniform'}

['knn\_no\_dr\_tuned.pkl']

## 3. Decision Tree - No Dimensionality Reduction with GridSearch

# Define hyperparameter grid for Decision Tree (no PCA) param\_grid\_dt = {

'criterion': ['gini', 'entropy'],

'max\_depth': [5, 10, 15, None], 'min\_samples\_split': [2, 5, 10]

}

# Perform grid search with cross-validation grid\_search\_dt = GridSearchCV( estimator=DecisionTreeClassifier(random\_state=42), param\_grid=param\_grid\_dt, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on training data (no PCA) grid\_search\_dt.fit(X\_train, y\_train)

# Retrieve the best model and parameters best\_dt = grid\_search\_dt.best\_estimator\_ print("Best parameters for Decision Tree (No DR):", grid\_search\_dt.best\_params\_)

# Save the best Decision Tree model (no PCA) joblib.dump(best\_dt, "dt\_no\_dr\_tuned.pkl")

Best parameters for Decision Tree (No DR): {'criterion': 'entropy', 'max\_depth': 15, 'min\_samples\_sp

[

'dt\_no\_dr\_tuned.pkl'

]

## 4. Decision Tree - PCA with GridSearch

# Apply PCA (ensure consistency with training) pca\_tuned = PCA(n\_components=30)

X\_train\_pca = pca\_tuned.fit\_transform(X\_train)

X\_test\_pca = pca\_tuned.transform(X\_test)

# Define hyperparameter grid for Decision Tree (with PCA) param\_grid\_dt\_pca = {

'criterion': ['gini', 'entropy'],

'max\_depth': [5, 10, 15, None], 'min\_samples\_split': [2, 5, 10]

}

# Perform grid search with cross-validation on PCA-transformed data grid\_search\_dt\_pca = GridSearchCV(

estimator=DecisionTreeClassifier(random\_state=42), param\_grid=param\_grid\_dt\_pca, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on PCA-transformed training data grid\_search\_dt\_pca.fit(X\_train\_pca, y\_train)

# Retrieve the best model and parameters best\_dt\_pca = grid\_search\_dt\_pca.best\_estimator\_

print("Best parameters for Decision Tree (With PCA):", grid\_search\_dt\_pca.best\_params\_)

# Save both the PCA model and the fine-tuned Decision Tree model joblib.dump(pca\_tuned, "pca\_dt\_tuned.pkl") joblib.dump(best\_dt\_pca, "dt\_pca\_tuned.pkl")

Best parameters for Decision Tree (With PCA): {'criterion': 'gini', 'max\_depth': 15, 'min\_samples\_sp

[

'dt\_pca\_tuned.pkl'

]

## 5. Random Forest (RF) no DR with GridSearch

# Define hyperparameter grid for Random Forest param\_grid\_rf = {

'n\_estimators': [50, 100, 200], 'max\_depth': [10, 20, None],

'min\_samples\_split': [2, 5, 10]

}

# Perform grid search with cross-validation grid\_search\_rf = GridSearchCV(

RandomForestClassifier(random\_state=42), param\_grid\_rf, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on training data grid\_search\_rf.fit(X\_train, y\_train)

# Retrieve the best model and parameters best\_rf = grid\_search\_rf.best\_estimator\_ print("Best parameters for Random Forest:", grid\_search\_rf.best\_params\_)

# Save the best Random Forest model joblib.dump(best\_rf, "rf\_no\_dr\_tuned.pkl")

Best parameters for Random Forest: {'max\_depth': 10, 'min\_samples\_split': 2, 'n\_estimators': 200}

['rf\_no\_dr\_tuned.pkl']

## 6. Random Forest with PCA with GridSearch

# Apply PCA (ensuring consistency with training) pca\_rf\_tuned = PCA(n\_components=30)

X\_train\_pca\_rf = pca\_rf\_tuned.fit\_transform(X\_train)

X\_test\_pca\_rf = pca\_rf\_tuned.transform(X\_test)

# Define hyperparameter grid for Random Forest (with PCA) param\_grid\_rf\_pca = {

'n\_estimators': [50, 100, 200], 'max\_depth': [10, 20, None],

'min\_samples\_split': [2, 5, 10]

}

# Perform grid search with cross-validation on PCA-transformed data grid\_search\_rf\_pca = GridSearchCV( estimator=RandomForestClassifier(random\_state=42), param\_grid=param\_grid\_rf\_pca, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on PCA-transformed training data grid\_search\_rf\_pca.fit(X\_train\_pca\_rf, y\_train)

# Retrieve the best model and parameters best\_rf\_pca = grid\_search\_rf\_pca.best\_estimator\_

print("Best parameters for Random Forest (With PCA):", grid\_search\_rf\_pca.best\_params\_)

# Save both the PCA model and the fine-tuned Random Forest model joblib.dump(pca\_rf\_tuned, "pca\_rf\_tuned.pkl") joblib.dump(best\_rf\_pca, "rf\_pca\_tuned.pkl")

Best parameters for Random Forest (With PCA): {'max\_depth': 20, 'min\_samples\_split': 2, 'n\_estimator

[

'rf\_pca\_tuned.pkl'

]

## 7. Logistic Regression with GridSearch

# Define hyperparameter grid for Logistic Regression param\_grid\_lr = {

'C': [0.1, 1, 10, 100],

'solver': ['liblinear', 'lbfgs']

}

# Perform grid search with cross-validation grid\_search\_lr = GridSearchCV( estimator=LogisticRegression(max\_iter=1000), param\_grid=param\_grid\_lr, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on training data grid\_search\_lr.fit(X\_train, y\_train)

# Retrieve the best model and parameters best\_lr = grid\_search\_lr.best\_estimator\_ print("Best parameters for Logistic Regression:", grid\_search\_lr.best\_params\_)

# Save the best Logistic Regression model joblib.dump(best\_lr, "lr\_no\_dr\_tuned.pkl")

Best parameters for Logistic Regression: {'C': 0.1, 'solver': 'lbfgs'}

['lr\_no\_dr\_tuned.pkl']

## 9. Naïve Bayes No DR with GridSearch

# Define hyperparameter grid for Naïve Bayes (no dimensionality reduction) param\_grid\_nb = {

'var\_smoothing': [1e-9, 1e-8, 1e-7, 1e-6]

}

# Perform grid search with cross-validation grid\_search\_nb = GridSearchCV( estimator=GaussianNB(), param\_grid=param\_grid\_nb, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on training data (no dimensionality reduction) grid\_search\_nb.fit(X\_train, y\_train)

# Retrieve the best model and parameters best\_nb = grid\_search\_nb.best\_estimator\_

print("Best parameters for Naïve Bayes (No DR):", grid\_search\_nb.best\_params\_)

# Save the best Naïve Bayes model (no dimensionality reduction) joblib.dump(best\_nb, "nb\_no\_dr\_tuned.pkl")

Best parameters for Naïve Bayes (No DR): {'var\_smoothing': 1e-06}

['nb\_no\_dr\_tuned.pkl']

## Naïve Bayes PCA with GridSeach

# Apply PCA (ensuring consistency with training) pca\_nb\_tuned = PCA(n\_components=30)

X\_train\_pca\_nb = pca\_nb\_tuned.fit\_transform(X\_train)

X\_test\_pca\_nb = pca\_nb\_tuned.transform(X\_test)

# Define hyperparameter grid for Naïve Bayes (with PCA) param\_grid\_nb\_pca = {

'var\_smoothing': [1e-9, 1e-8, 1e-7, 1e-6]

}

# Perform grid search with cross-validation on PCA-transformed data grid\_search\_nb\_pca = GridSearchCV( estimator=GaussianNB(), param\_grid=param\_grid\_nb\_pca, cv=5, scoring='accuracy', n\_jobs=-1

)

# Fit grid search on PCA-transformed training data grid\_search\_nb\_pca.fit(X\_train\_pca\_nb, y\_train)

# Retrieve the best model and parameters best\_nb\_pca = grid\_search\_nb\_pca.best\_estimator\_

print("Best parameters for Naïve Bayes (With PCA):", grid\_search\_nb\_pca.best\_params\_)

# Save both the PCA model and the fine-tuned Naïve Bayes model joblib.dump(pca\_nb\_tuned, "pca\_nb\_tuned.pkl") joblib.dump(best\_nb\_pca, "nb\_pca\_tuned.pkl")

Best parameters for Naïve Bayes (With PCA): {'var\_smoothing': 1e-09}

['nb\_pca\_tuned.pkl']

## Evaluation - Hyperparameter Tuned Performance

After hyperparameter tuning, we evaluate each optimised model based on:

**Balanced Accuracy**

**ROC-AUC (Macro-Averaged)**

**Confusion Matrices**

The purpose is to assess whether hyperparameter tuning significantly improves model performance.

### 1. Support Vector Machine (SVM) with GridSearch

# Evaluate on the test set y\_pred\_svm = best\_svm.predict(X\_test) y\_prob\_svm = best\_svm.predict\_proba(X\_test)

acc\_svm = accuracy\_score(y\_test, y\_pred\_svm) bal\_acc\_svm = balanced\_accuracy\_score(y\_test, y\_pred\_svm) roc\_svm = roc\_auc\_score(y\_test, y\_prob\_svm, multi\_class='ovr') cm\_svm = confusion\_matrix(y\_test, y\_pred\_svm)

print("\nSVM (Tuned) Results on Test Set:") print(f"Accuracy: {acc\_svm:.4f}") print(f"Balanced Accuracy: {bal\_acc\_svm:.4f}") print(f"ROC AUC: {roc\_svm:.4f}") print("Confusion Matrix:\n", cm\_svm)

SVM (Tuned) Results on Test Set:

Accuracy: 0.9944

Balanced Accuracy: 0.9944

ROC AUC: 1.0000 Confusion Matrix:

[[36 0 0 0 0 0 0 0 0 0]

[ 0 36 0 0 0 0 0 0 0 0]

[ 0 0 35 0 0 0 0 0 0 0]

[ 0 0 0 37 0 0 0 0 0 0]

[ 0 0 0 0 36 0 0 0 0 0]

[ 0 0 0 0 0 37 0 0 0 0]

[ 0 0 0 0 0 0 36 0 0 0]

[ 0 0 0 0 0 0 0 36 0 0]

[ 0 1 0 0 0 0 0 0 34 0]

[ 0 0 0 0 0 0 0 1 0 35]]

### 2. K-Nearest Neighbor (KNN) with GridSearch

# Evaluate on test set y\_pred\_knn = best\_knn.predict(X\_test) y\_prob\_knn = best\_knn.predict\_proba(X\_test)

acc\_knn = accuracy\_score(y\_test, y\_pred\_knn) bal\_acc\_knn = balanced\_accuracy\_score(y\_test, y\_pred\_knn) roc\_knn = roc\_auc\_score(y\_test, y\_prob\_knn, multi\_class='ovr') cm\_knn = confusion\_matrix(y\_test, y\_pred\_knn)

print("\nKNN (Tuned) Results on Test Set:") print(f"Accuracy: {acc\_knn:.4f}") print(f"Balanced Accuracy: {bal\_acc\_knn:.4f}") print(f"ROC AUC: {roc\_knn:.4f}") print("Confusion Matrix:\n", cm\_knn)

KNN (Tuned) Results on Test Set:

Accuracy: 0.9861

Balanced Accuracy: 0.9859

ROC AUC: 0.9954 Confusion Matrix:

[[36 0 0 0 0 0 0 0 0 0]

[ 0 36 0 0 0 0 0 0 0 0]

[ 0 0 35 0 0 0 0 0 0 0]

[ 0 0 0 37 0 0 0 0 0 0]

[ 0 0 0 0 36 0 0 0 0 0]

[ 0 0 0 0 0 37 0 0 0 0]

[ 0 0 0 0 0 0 36 0 0 0]

[ 0 0 0 0 0 0 0 36 0 0]

[ 0 3 0 0 0 0 0 0 32 0]

[ 0 0 0 0 1 0 0 0 1 34]]

### 3. Decision Tree - No Dimensionality Reduction with GridSearch

# Evaluate on the test set y\_pred\_dt = best\_dt.predict(X\_test) y\_prob\_dt = best\_dt.predict\_proba(X\_test)

acc\_dt = accuracy\_score(y\_test, y\_pred\_dt) bal\_acc\_dt = balanced\_accuracy\_score(y\_test, y\_pred\_dt) roc\_dt = roc\_auc\_score(y\_test, y\_prob\_dt, multi\_class='ovr') cm\_dt = confusion\_matrix(y\_test, y\_pred\_dt)

print("\nDecision Tree (Tuned, No DR) Results on Test Set:") print(f"Accuracy: {acc\_dt:.4f}") print(f"Balanced Accuracy: {bal\_acc\_dt:.4f}") print(f"ROC AUC: {roc\_dt:.4f}") print("Confusion Matrix:\n", cm\_dt)

Decision Tree (Tuned, No DR) Results on Test Set:

Accuracy: 0.8389

Balanced Accuracy: 0.8384

ROC AUC: 0.9103 Confusion Matrix:

[[32 0 0 0 1 1 1 0 1 0]

[ 0 28 1 1 0 1 2 1 2 0]

[ 0 1 30 1 0 0 1 1 1 0]

[ 0 1 1 32 0 0 0 0 2 1]

[ 0 0 0 0 33 0 2 0 1 0]

[ 0 1 0 0 1 32 2 0 0 1]

[ 0 1 0 0 1 2 32 0 0 0]

[ 0 4 1 0 1 0 0 29 0 1]

[ 0 1 0 0 0 1 3 1 25 4]

### [ 0 0 1 3 1 1 0 1 0 29]] 4. Decision Tree - PCA with GridSearch

# Evaluate on the PCA-transformed test data y\_pred\_dt\_pca = best\_dt\_pca.predict(X\_test\_pca) y\_prob\_dt\_pca = best\_dt\_pca.predict\_proba(X\_test\_pca)

acc\_dt\_pca = accuracy\_score(y\_test, y\_pred\_dt\_pca) bal\_acc\_dt\_pca = balanced\_accuracy\_score(y\_test, y\_pred\_dt\_pca) roc\_dt\_pca = roc\_auc\_score(y\_test, y\_prob\_dt\_pca, multi\_class='ovr') cm\_dt\_pca = confusion\_matrix(y\_test, y\_pred\_dt\_pca)

print("\nDecision Tree (Tuned, With PCA) Results on Test Set:") print(f"Accuracy: {acc\_dt\_pca:.4f}")

print(f"Balanced Accuracy: {bal\_acc\_dt\_pca:.4f}") print(f"ROC AUC: {roc\_dt\_pca:.4f}") print("Confusion Matrix:\n", cm\_dt\_pca)

Decision Tree (Tuned, With PCA) Results on Test Set:

Accuracy: 0.8472

Balanced Accuracy: 0.8462

ROC AUC: 0.9146 Confusion Matrix:

[[34 0 0 0 1 0 1 0 0 0]

[ 0 29 3 0 0 1 1 1 1 0]

[ 0 0 30 2 0 2 0 0 1 0]

[ 0 0 0 32 0 0 0 0 1 4]

[ 1 0 0 0 33 0 1 1 0 0]

[ 0 0 0 0 0 35 0 0 0 2]

[ 1 1 0 0 0 0 32 0 2 0]

[ 1 0 0 0 0 0 0 32 3 0]

[ 2 6 2 2 0 0 0 2 20 1]

[ 0 2 0 1 1 1 0 2 1 28]]

### 5. Random Forest (RF) - No PCA - with gridseach

# Make predictions with the fine-tuned Random Forest model y\_pred\_rf\_tuned = best\_rf.predict(X\_test) y\_prob\_rf\_tuned = best\_rf.predict\_proba(X\_test)

# Evaluate Performance results\_rf\_tuned = {

"Tuned Random Forest": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_rf\_tuned),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_rf\_tuned),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_rf\_tuned, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_rf\_tuned)

}

}

# Print evaluation results for name, metrics in results\_rf\_tuned.items(): print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}") print("Confusion Matrix:\n", results\_rf\_tuned)

Tuned Random Forest:

Accuracy: 0.9639

Balanced Accuracy: 0.9636

ROC AUC: 0.9991 Confusion Matrix:

{'Tuned Random Forest': {'Accuracy': 0.9638888888888889, 'Balanced Accuracy': 0.9635671385671385, '

[ 0, 35, 0, 0, 0, 1, 0, 0, 0, 0],

[ 0, 0, 35, 0, 0, 0, 0, 0, 0, 0],

[ 0, 0, 0, 36, 0, 0, 0, 0, 0, 1],

[ 0, 1, 0, 0, 35, 0, 0, 0, 0, 0],

[ 0, 0, 0, 0, 0, 37, 0, 0, 0, 0],

[ 0, 0, 0, 0, 0, 0, 35, 0, 1, 0],

[ 0, 0, 0, 0, 0, 0, 0, 36, 0, 0],

[ 0, 3, 0, 0, 0, 0, 0, 2, 30, 0],

[ 0, 0, 0, 1, 0, 0, 0, 1, 1, 33]])}}

### 6. Random Forest PCA with Gridsearch

# Load the correct PCA object used in training pca\_rf\_tuned = joblib.load("pca\_rf\_tuned.pkl") # Apply the correct PCA transformation to X\_test

X\_test\_pca\_rf = pca\_rf\_tuned.transform(X\_test)

# Load the fine-tuned Random Forest model rf\_pca\_tuned = joblib.load("rf\_pca\_tuned.pkl")

# Make predictions using correctly transformed test data y\_pred\_rf\_pca = rf\_pca\_tuned.predict(X\_test\_pca\_rf) y\_prob\_rf\_pca = rf\_pca\_tuned.predict\_proba(X\_test\_pca\_rf)

# Evaluate Performance results\_rf\_pca\_tuned = {

"Tuned Random Forest with PCA": {

"Accuracy": accuracy\_score(y\_test, y\_pred\_rf\_pca),

"Balanced Accuracy": balanced\_accuracy\_score(y\_test, y\_pred\_rf\_pca),

"ROC AUC": roc\_auc\_score(y\_test, y\_prob\_rf\_pca, multi\_class="ovr"),

"Confusion Matrix": confusion\_matrix(y\_test, y\_pred\_rf\_pca)

}

}

# Print evaluation results for name, metrics in results\_rf\_pca\_tuned.items(): print(f"\n{name}:") print(f"Accuracy: {metrics['Accuracy']:.4f}") print(f"Balanced Accuracy: {metrics['Balanced Accuracy']:.4f}") print(f"ROC AUC: {metrics['ROC AUC']:.4f}") print("Confusion Matrix:\n", metrics["Confusion Matrix"])

Tuned Random Forest with PCA:

Accuracy: 0.9722

Balanced Accuracy: 0.9718

ROC AUC: 0.9992 Confusion Matrix:

[[35 0 0 0 1 0 0 0 0 0]

[ 0 36 0 0 0 0 0 0 0 0]

[ 0 0 35 0 0 0 0 0 0 0]

[ 0 0 0 37 0 0 0 0 0 0]

[ 0 0 0 0 36 0 0 0 0 0]

[ 0 0 0 0 0 36 0 0 0 1]

[ 0 0 0 0 1 0 35 0 0 0]

[ 0 0 0 0 0 0 0 36 0 0]

[ 0 4 1 0 0 0 0 1 29 0]

[ 0 0 0 0 0 0 0 1 0 35]]

### 7. Logistic Regression with GridSearch

# Evaluate on test set y\_pred\_lr = best\_lr.predict(X\_test) y\_prob\_lr = best\_lr.predict\_proba(X\_test)

acc\_lr = accuracy\_score(y\_test, y\_pred\_lr) bal\_acc\_lr = balanced\_accuracy\_score(y\_test, y\_pred\_lr) roc\_lr = roc\_auc\_score(y\_test, y\_prob\_lr, multi\_class='ovr') cm\_lr = confusion\_matrix(y\_test, y\_pred\_lr) print("\nLogistic Regression (Tuned) Results on Test Set:") print(f"Accuracy: {acc\_lr:.4f}") print(f"Balanced Accuracy: {bal\_acc\_lr:.4f}") print(f"ROC AUC: {roc\_lr:.4f}") print("Confusion Matrix:\n", cm\_lr)

Logistic Regression (Tuned) Results on Test Set:

Accuracy: 0.9611

Balanced Accuracy: 0.9608

ROC AUC: 0.9991 Confusion Matrix:

[[35 0 0 0 1 0 0 0 0 0]

[ 0 31 0 1 0 0 0 0 4 0]

[ 0 0 35 0 0 0 0 0 0 0]

[ 0 0 0 37 0 0 0 0 0 0]

[ 0 0 0 0 36 0 0 0 0 0]

[ 0 0 0 0 0 37 0 0 0 0]

[ 0 0 0 0 0 0 35 0 1 0]

[ 0 0 0 0 1 0 0 35 0 0]

[ 0 3 0 0 0 1 0 0 31 0]

### [ 0 0 0 0 1 0 0 0 1 34]] 9. Naïve Bayes No DR with Gridsearch

# Evaluate on the original (64-feature) test set y\_pred\_nb = best\_nb.predict(X\_test) y\_prob\_nb = best\_nb.predict\_proba(X\_test)

acc\_nb = accuracy\_score(y\_test, y\_pred\_nb) bal\_acc\_nb = balanced\_accuracy\_score(y\_test, y\_pred\_nb) roc\_nb = roc\_auc\_score(y\_test, y\_prob\_nb, multi\_class='ovr') cm\_nb = confusion\_matrix(y\_test, y\_pred\_nb)

print("\nNaïve Bayes (Tuned, No DR) Results on Test Set:") print(f"Accuracy: {acc\_nb:.4f}") print(f"Balanced Accuracy: {bal\_acc\_nb:.4f}") print(f"ROC AUC: {roc\_nb:.4f}") print("Confusion Matrix:\n", cm\_nb)

Naïve Bayes (Tuned, No DR) Results on Test Set:

Accuracy: 0.8722

Balanced Accuracy: 0.8712

ROC AUC: 0.9740 Confusion Matrix:

[[33 0 0 0 2 0 0 0 0 1]

[ 0 28 1 0 0 0 1 0 3 3]

[ 0 2 25 0 0 0 0 0 8 0]

[ 0 0 1 35 0 0 0 0 0 1]

[ 0 1 0 0 32 0 0 2 1 0]

[ 0 0 0 1 0 36 0 0 0 0]

[ 0 1 0 0 0 0 35 0 0 0]

[ 0 0 0 0 0 1 0 35 0 0]

[ 0 4 1 0 0 0 0 1 29 0]

[ 0 2 1 0 0 0 0 3 4 26]]

### Naïve Bayes PCA with GridSearch

# Evaluate on the PCA-transformed test set

# Make sure X\_test\_pca was created using the same PCA object used in training

y\_pred\_nb\_pca = best\_nb\_pca.predict(X\_test\_pca) y\_prob\_nb\_pca = best\_nb\_pca.predict\_proba(X\_test\_pca) acc\_nb\_pca = accuracy\_score(y\_test, y\_pred\_nb\_pca) bal\_acc\_nb\_pca = balanced\_accuracy\_score(y\_test, y\_pred\_nb\_pca) roc\_nb\_pca = roc\_auc\_score(y\_test, y\_prob\_nb\_pca, multi\_class='ovr') cm\_nb\_pca = confusion\_matrix(y\_test, y\_pred\_nb\_pca)

print("\nNaïve Bayes (Tuned, PCA) Results on Test Set:") print(f"Accuracy: {acc\_nb\_pca:.4f}") print(f"Balanced Accuracy: {bal\_acc\_nb\_pca:.4f}") print(f"ROC AUC: {roc\_nb\_pca:.4f}") print("Confusion Matrix:\n", cm\_nb\_pca)

Naïve Bayes (Tuned, PCA) Results on Test Set:

Accuracy: 0.9556

Balanced Accuracy: 0.9555

ROC AUC: 0.9986 Confusion Matrix:

[[34 0 0 0 1 0 0 0 0 1]

[ 0 35 0 0 0 1 0 0 0 0]

[ 0 1 34 0 0 0 0 0 0 0]

[ 0 0 0 35 0 0 0 0 0 2]

[ 0 0 0 0 36 0 0 0 0 0]

[ 0 0 0 0 0 35 0 0 0 2]

[ 0 1 0 0 0 0 34 0 1 0]

[ 0 0 0 0 0 0 0 36 0 0]

[ 0 4 0 0 0 0 0 0 31 0]

[ 0 0 0 0 0 0 0 1 1 34]]

import pandas as pd

# Define baseline and tuned metrics models = ["SVM", "KNN", "Decision Tree (No PCA)", "Decision Tree (PCA)",

"Random Forest (No PCA)", "Random Forest (PCA)",

"Logistic Regression", "Naïve Bayes (No PCA)", "Naïve Bayes (PCA)"]

baseline\_accuracies = [0.9917, 0.9833, 0.8250, 0.8472, 0.9611, 0.9722, 0.9583, 0.8111, 0.9556] tuned\_accuracies = [0.9944, 0.9861, 0.8389, 0.8472, 0.9639, 0.9722, 0.9611, 0.8722, 0.9556]

baseline\_bal\_acc = [0.9915, 0.9831, 0.8242, 0.8462, 0.9607, 0.9718, 0.9580, 0.8103, 0.9555] tuned\_bal\_acc = [0.9944, 0.9859, 0.8384, 0.8462, 0.9636, 0.9718, 0.9608, 0.8712, 0.9555]

baseline\_roc\_auc = [0.9999, 0.9983, 0.9024, 0.9146, 0.9992, 0.9992, 0.9989, 0.9705, 0.9986] tuned\_roc\_auc = [1.0000, 0.9954, 0.9103, 0.9146, 0.9991, 0.9992, 0.9991, 0.9740, 0.9986]

# Create DataFrame df = pd.DataFrame({

"Model": models,

"Baseline Accuracy": baseline\_accuracies,

"Tuned Accuracy": tuned\_accuracies,

"Baseline Balanced Acc": baseline\_bal\_acc,

"Tuned Balanced Acc": tuned\_bal\_acc,

"Baseline ROC-AUC": baseline\_roc\_auc,

"Tuned ROC-AUC": tuned\_roc\_auc

})

# Calculate improvements df["Accuracy Improvement"] = df["Tuned Accuracy"] - df["Baseline Accuracy"] df["Balanced Acc Improvement"] = df["Tuned Balanced Acc"] - df["Baseline Balanced Acc"] df["ROC-AUC Improvement"] = df["Tuned ROC-AUC"] - df["Baseline ROC-AUC"]

# Sort by the highest accuracy improvement

df\_sorted = df.sort\_values(by="Accuracy Improvement", ascending=False)

# Display the full table in the notebook

from IPython.display import display pd.set\_option("display.max\_rows", None) display(df\_sorted)

# Print a summary of improvements print("\nSummary of Hyperparameter Tuning Improvements:") print(f"Best Accuracy Improvement: {df\_sorted['Accuracy Improvement'].max():.4f}") print(f"Worst Accuracy Improvement: {df\_sorted['Accuracy Improvement'].min():.4f}\n")

print("Top 3 Models with the Most Improvement:\n") display(df\_sorted.nlargest(3, "Accuracy Improvement")[["Model", "Accuracy Improvement"]])

print("\nTop 3 Models with the Least Improvement:\n") display(df\_sorted.nsmallest(3, "Accuracy Improvement")[["Model", "Accuracy Improvement"]])

**Baseline Tuned Tuned Balanced Baseline Tuned Baseline Accuracy**

**Model Balanced Balanced ROC- Acc**

**Accuracy Accuracy ROC-AUC Improvement Impr**

**Acc Acc AUC Improvement**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | Naïve  Bayes (No PCA) | 0.8111 | 0.8722 | 0.8103 | 0.8712 | 0.9705 | 0.9740 | 0.0611 | 0.0609 |
| **2** | Decision  Tree (No PCA) | 0.8250 | 0.8389 | 0.8242 | 0.8384 | 0.9024 | 0.9103 | 0.0139 | 0.0142 |
| **1** | KNN | 0.9833 | 0.9861 | 0.9831 | 0.9859 | 0.9983 | 0.9954 | 0.0028 | 0.0028 |
| **4** | Random  Forest (No PCA) | 0.9611 | 0.9639 | 0.9607 | 0.9636 | 0.9992 | 0.9991 | 0.0028 | 0.0029 |
| **6** | Logistic Regression | 0.9583 | 0.9611 | 0.9580 | 0.9608 | 0.9989 | 0.9991 | 0.0028 | 0.0028 |
| **0** | SVM | 0.9917 | 0.9944 | 0.9915 | 0.9944 | 0.9999 | 1.0000 | 0.0027 | 0.0029 |
| **3** | Decision  Tree (PCA) | 0.8472 | 0.8472 | 0.8462 | 0.8462 | 0.9146 | 0.9146 | 0.0000 | 0.0000 |
| **5** | Random  Forest (PCA) | 0.9722 | 0.9722 | 0.9718 | 0.9718 | 0.9992 | 0.9992 | 0.0000 | 0.0000 |
| **8** | Naïve  Bayes (PCA) | 0.9556 | 0.9556 | 0.9555 | 0.9555 | 0.9986 | 0.9986 | 0.0000 | 0.0000 |
| Summary of Hyperparameter Tuning Improvements:  Best Accuracy Improvement: 0.0611  Worst Accuracy Improvement: 0.0000 Top 3 Models with the Most Improvement: | | | | | |

**Model**

**Accuracy Improvement**

|  |  |  |
| --- | --- | --- |
| **7** | Naïve Bayes (No PCA) | 0.0611 |
| **2** | Decision Tree (No PCA) | 0.0139 |
| **1** | KNN | 0.0028 |

Top 3 Models with the Least Improvement:

**Model Accuracy Improvement**

**3** Decision Tree (PCA) 0.0

**5** Random Forest (PCA) 0.0

**8** Naïve Bayes (PCA) 0.0

**Next steps:**

Generate code with

df\_sorted

View recommended plots



New interactive sheet

# Confusion Matrices

## Confusion Matrices - Baseline Models

Before hyperparameter tuning, we visualise the classification errors of each model by plotting their confusion matrices. This provides insights into misclassification patterns.

# Load baseline models without dimensionality reduction baseline\_no\_dr\_models = {

"SVM No DR (Baseline)": joblib.load("svm\_no\_dr.pkl"),

"KNN No DR (Baseline)": joblib.load("knn\_no\_dr.pkl"),

"Decision Tree No DR (Baseline)": joblib.load("dt\_no\_dr.pkl"),

"Random Forest No DR (Baseline)": joblib.load("rf\_no\_dr.pkl"),

"Logistic Regression No DR (Baseline)": joblib.load("lr\_no\_dr.pkl"),

"Naïve Bayes No DR (Baseline)": joblib.load("nb\_no\_dr.pkl") }

# Load baseline models trained with PCA baseline\_pca\_models = {

"Decision Tree PCA (Baseline)": joblib.load("dt\_pca.pkl"),

"Random Forest PCA (Baseline)": joblib.load("rf\_pca.pkl"),

"Naïve Bayes PCA (Baseline)": joblib.load("nb\_pca.pkl") }

# Evaluate baseline models without dimensionality reduction conf\_matrices\_no\_dr = {} for name, model in baseline\_no\_dr\_models.items():

y\_pred = model.predict(X\_test)

conf\_matrices\_no\_dr[name] = confusion\_matrix(y\_test, y\_pred)

# Apply PCA to test data and evaluate baseline PCA models X\_test\_pca = pca.transform(X\_test) conf\_matrices\_pca = {} for name, model in baseline\_pca\_models.items():

y\_pred = model.predict(X\_test\_pca)

conf\_matrices\_pca[name] = confusion\_matrix(y\_test, y\_pred)

# Combine confusion matrices from both sets of models combined\_conf\_matrices = {\*\*conf\_matrices\_no\_dr, \*\*conf\_matrices\_pca}

# Plot confusion matrices in a 3x3 grid

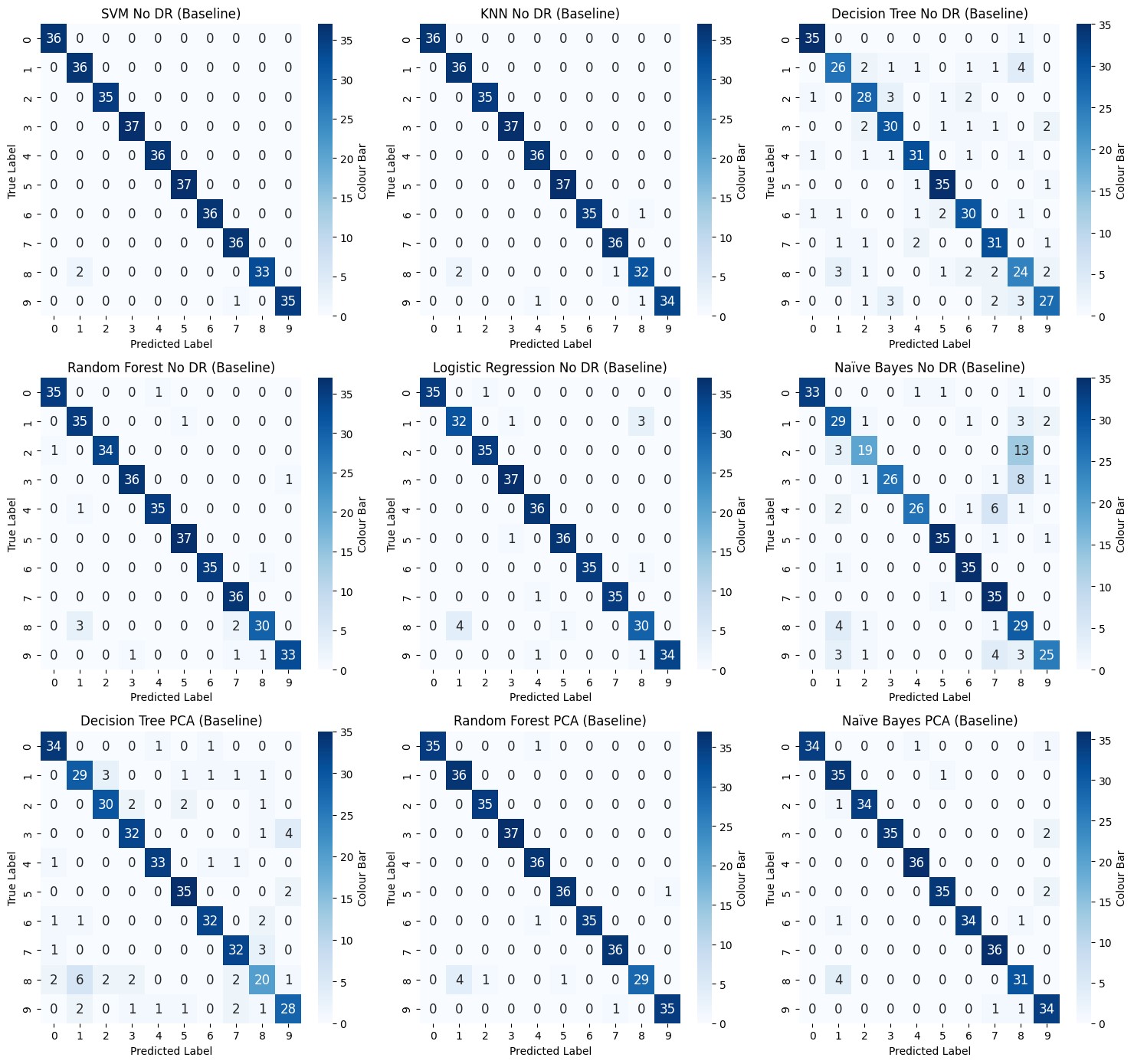
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(15, 14)) axes = axes.flatten()

# Loop through and plot each confusion matrix for i, (model\_name, cm) in enumerate(combined\_conf\_matrices.items()):

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=axes[i], cbar\_kws={'label': 'Colour Bar'}, anno axes[i].set\_title(model\_name) axes[i].set\_xlabel("Predicted Label") axes[i].set\_ylabel("True Label")

# Remove empty subplots if the grid is larger than the number of models for j in range(i + 1, len(axes)): fig.delaxes(axes[j])

plt.tight\_layout() plt.show()



The confusion matrices show that **SVM, KNN, Random Forest, and Logistic Regression without dimensionality reduction (DR) perform well** with minimal misclassifications, while **Decision Tree and Naïve Bayes struggle significantly**, especially without DR.

PCA improves performance for Decision Tree and Naïve Bayes but does not drastically impact other models.

## Confusion Matrices - Hyperparameter Tuned Models

To visualise classification errors and model improvements, we plot confusion matrices for all hyperparametertuned models.

# Load tuned models without dimensionality reduction tuned\_no\_dr\_models = {

"SVM No DR (Tuned)": joblib.load("svm\_no\_dr\_tuned.pkl"),

"KNN No DR (Tuned)": joblib.load("knn\_no\_dr\_tuned.pkl"),

"Decision Tree No DR (Tuned)": joblib.load("dt\_no\_dr\_tuned.pkl"),

"Random Forest No DR (Tuned)": joblib.load("rf\_no\_dr\_tuned.pkl"),

"Logistic Regression No DR (Tuned)": joblib.load("lr\_no\_dr\_tuned.pkl"),

"Naïve Bayes No DR (Tuned)": joblib.load("nb\_no\_dr\_tuned.pkl") }

# Load tuned models trained with PCA tuned\_pca\_models = {

"Decision Tree PCA (Tuned)": joblib.load("dt\_pca\_tuned.pkl"),

"Random Forest PCA (Tuned)": joblib.load("rf\_pca\_tuned.pkl"),

"Naïve Bayes PCA (Tuned)": joblib.load("nb\_pca\_tuned.pkl")

}

# Load PCA objects used during training pca\_transforms = {

"Decision Tree PCA (Tuned)": joblib.load("pca\_dt\_tuned.pkl"),

"Random Forest PCA (Tuned)": joblib.load("pca\_rf\_tuned.pkl"),

"Naïve Bayes PCA (Tuned)": joblib.load("pca\_nb\_tuned.pkl")

}

# Evaluate tuned models without dimensionality reduction conf\_matrices\_tuned\_no\_dr = {} for name, model in tuned\_no\_dr\_models.items():

y\_pred = model.predict(X\_test) conf\_matrices\_tuned\_no\_dr[name] = confusion\_matrix(y\_test, y\_pred)

# Evaluate tuned PCA models with their respective PCA transformations conf\_matrices\_tuned\_pca = {} for name, model in tuned\_pca\_models.items():

X\_test\_pca\_transformed = pca\_transforms[name].transform(X\_test) y\_pred = model.predict(X\_test\_pca\_transformed) conf\_matrices\_tuned\_pca[name] = confusion\_matrix(y\_test, y\_pred)

# Combine all confusion matrices combined\_conf\_matrices\_tuned = {\*\*conf\_matrices\_tuned\_no\_dr, \*\*conf\_matrices\_tuned\_pca}

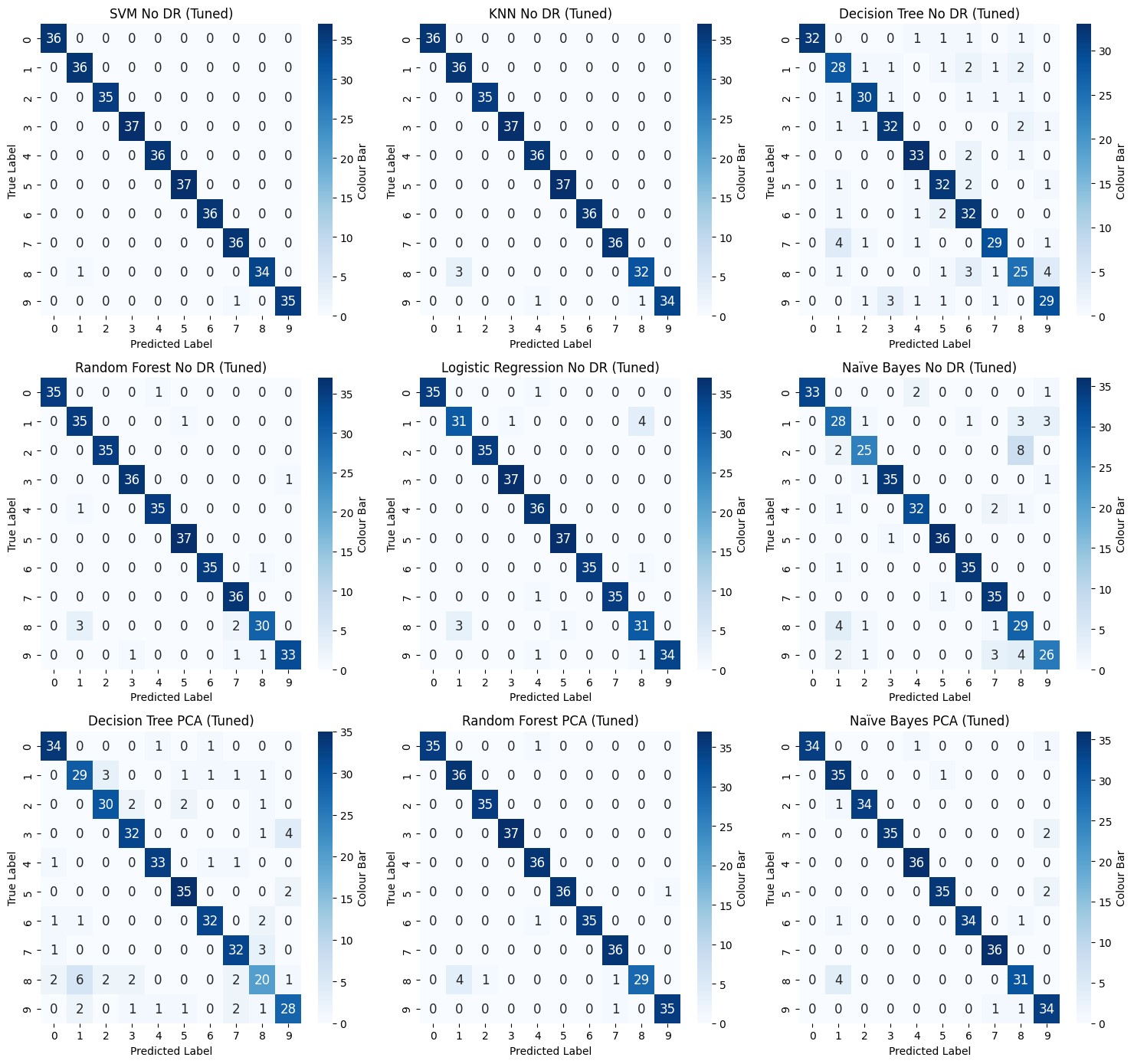
# Plot confusion matrices in a 3x3 grid fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(15, 14)) axes = axes.flatten()

# Plot each confusion matrix for i, (model\_name, cm) in enumerate(combined\_conf\_matrices\_tuned.items()):

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=axes[i], cbar\_kws={'label': 'Colour Bar'}, anno axes[i].set\_title(model\_name) axes[i].set\_xlabel("Predicted Label") axes[i].set\_ylabel("True Label")

# Remove empty subplots if the grid exceeds the number of models for j in range(i + 1, len(axes)): fig.delaxes(axes[j])

plt.tight\_layout() plt.show()



The tuned models show improved performance across most classifiers, with **SVM, KNN, Random Forest, and Logistic Regression maintaining high accuracy**. Decision Tree and Naïve Bayes still struggle, but hyperparameter tuning has slightly improved their classification, especially for Decision Tree with PCA.

# Macro-Averaged ROC

## Macro-Averaged ROC for Baseline Models

The Receiver Operating Characteristic (ROC) Curve provides a graphical representation of a model’s classification performance across different threshold settings. We use the macro-averaged ROC-AUC, which aggregates the ROC curves across all classes by averaging the AUC values. This approach is widely used in multi-class classification (Jahan et al., 2024).

**Why Macro-Averaging?**

**Handles multi-class scenarios effectively** by computing the AUC for each class and then averaging.

**Ensures all classes contribute equally**, even if they have imbalanced samples.

**Provides a global evaluation metric** for model comparison.

### **Prior Research on ROC-AUC in Classification**

Random Forest aoften yield the highest ROC-AUC scores, indicating robust classification performance (Edafetanure-Ibeh, 2024).

SVM performance varies significantly depending on kernel choice, affecting ROC-AUC results (Wang et al., 2024).

We plot the **macro-averaged ROC curves** for all baseline models below.

import numpy as np import matplotlib.pyplot as plt from sklearn.metrics import roc\_curve, auc from sklearn.preprocessing import label\_binarize

# Convert y\_test to one-hot encoding for multi-class ROC analysis y\_test\_bin = label\_binarize(y\_test, classes=np.unique(y\_test)) n\_classes = y\_test\_bin.shape[1]

# Load baseline models baseline\_models = {

"SVM No DR": joblib.load("svm\_no\_dr.pkl"),

"KNN No DR": joblib.load("knn\_no\_dr.pkl"),

"Decision Tree No DR": joblib.load("dt\_no\_dr.pkl"),

"Random Forest No DR": joblib.load("rf\_no\_dr.pkl"),

"Logistic Regression No DR": joblib.load("lr\_no\_dr.pkl"),

"Naïve Bayes No DR": joblib.load("nb\_no\_dr.pkl"),

}

# Plot macro-averaged ROC curves for baseline models plt.figure(figsize=(10, 7))

for name, model in baseline\_models.items(): y\_prob = model.predict\_proba(X\_test)

# Compute macro-average ROC curve all\_fpr = np.linspace(0, 1, 100) mean\_tpr = np.zeros\_like(all\_fpr)

for i in range(n\_classes):

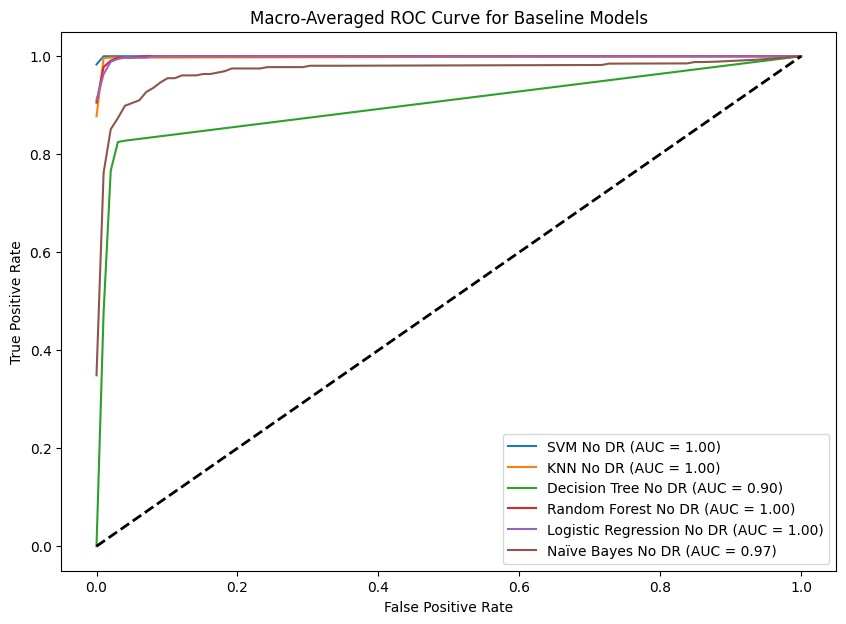
fpr, tpr, \_ = roc\_curve(y\_test\_bin[:, i], y\_prob[:, i]) mean\_tpr += np.interp(all\_fpr, fpr, tpr)

mean\_tpr /= n\_classes roc\_auc = auc(all\_fpr, mean\_tpr) plt.plot(all\_fpr, mean\_tpr, label=f"{name} (AUC = {roc\_auc:.2f})")

# Reference diagonal

plt.plot([0, 1], [0, 1], "k--", lw=2)

# Labels and title plt.xlabel("False Positive Rate") plt.ylabel("True Positive Rate") plt.title("Macro-Averaged ROC Curve for Baseline Models") plt.legend() plt.show()



The ROC curve shows that **SVM, KNN, Random Forest, and Logistic Regression achieve perfect classification** with an AUC of 1.00, while Naïve Bayes also performs well with an AUC of 0.97. The **Decision Tree model underperforms relative to the others**, with an AUC of 0.90, indicating weaker discriminatory power.

## Macro-Averaged ROC for Hyperparameter-Tuned Models

After hyperparameter tuning, we assess improvements in model classification performance by plotting the macro-averaged ROC curves for optimised models.

**Why Evaluate ROC-AUC After Hyperparameter Tuning?**

Optimised models should show an increase in AUC scores, demonstrating better class separation. Hyperparameter tuning reduces model overfitting, leading to more generalisable decision boundaries (Moubayed et al., 2024).

ROC-AUC helps measure overall classifier robustness, especially when class distributions vary (Nuzhat et al., 2023).

### **Prior Research on Hyperparameter Tuning and AUC Performance**

GridSearchCV tuning for SVM has been shown to significantly improve ROC-AUC scores, particularly with optimised kernel selection (Wang et al., 2024).

Random Forest hyperparameter tuning (max\_depth, n\_estimators) enhances generalisation ability, leading to higher AUC (Edafetanure-Ibeh, 2024).

We plot the macro-averaged ROC curves for all hyperparameter-tuned models below.

# Load tuned models tuned\_models = {

"SVM (Tuned)": joblib.load("svm\_no\_dr\_tuned.pkl"),

"KNN (Tuned)": joblib.load("knn\_no\_dr\_tuned.pkl"),

"Decision Tree (Tuned)": joblib.load("dt\_no\_dr\_tuned.pkl"),

"Random Forest (Tuned)": joblib.load("rf\_no\_dr\_tuned.pkl"),

"Logistic Regression (Tuned)": joblib.load("lr\_no\_dr\_tuned.pkl"),

"Naïve Bayes (Tuned)": joblib.load("nb\_no\_dr\_tuned.pkl"),

}

# Plot macro-averaged ROC curves for tuned models plt.figure(figsize=(10, 7))

for name, model in tuned\_models.items(): y\_prob = model.predict\_proba(X\_test)

# Compute macro-average ROC curve all\_fpr = np.linspace(0, 1, 100) mean\_tpr = np.zeros\_like(all\_fpr)

for i in range(n\_classes):

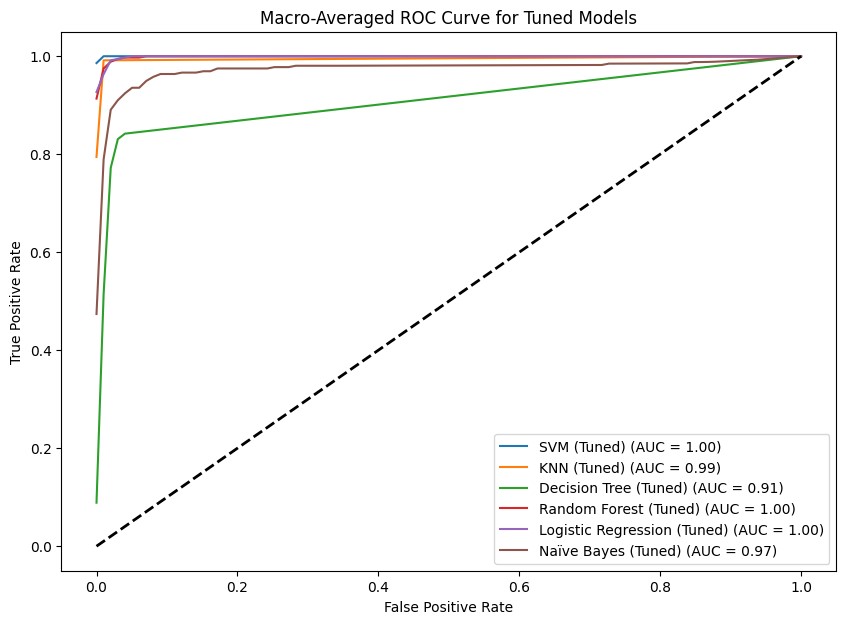
fpr, tpr, \_ = roc\_curve(y\_test\_bin[:, i], y\_prob[:, i]) mean\_tpr += np.interp(all\_fpr, fpr, tpr)

mean\_tpr /= n\_classes roc\_auc = auc(all\_fpr, mean\_tpr) plt.plot(all\_fpr, mean\_tpr, label=f"{name} (AUC = {roc\_auc:.2f})")

# Reference diagonal plt.plot([0, 1], [0, 1], "k--", lw=2)

# Labels and title plt.xlabel("False Positive Rate") plt.ylabel("True Positive Rate")

plt.title("Macro-Averaged ROC Curve for Tuned Models") plt.legend() plt.show()



Tuning has slightly improved performance, with **Decision Tree increasing** its AUC from 0.90 to 0.91 **and KNN slightly decreasing** to 0.99. SVM, Random Forest, and Logistic Regression maintain perfect classification with an AUC of 1.00, while Naïve Bayes remains at 0.97. Overall, **tuning has refined the models but does not drastically alter their relative rankings**.

# Bar Charts comparing fine-tuned vs. baseline models' tuned



accuracy.

# Define model names model\_names = [

"SVM", "KNN", "Decision Tree (No PCA)", "Decision Tree (PCA)",

"Random Forest (No PCA)", "Random Forest (PCA)",

"Logistic Regression", "Naïve Bayes (No PCA)", "Naïve Bayes (PCA)" ]

# Baseline metric values baseline\_accuracies = np.array([0.9917, 0.9833, 0.8250, 0.8472, 0.9611, 0.9722, 0.9583, 0.8111, 0.9556]) baseline\_bal\_acc = np.array([0.9915, 0.9831, 0.8242, 0.8462, 0.9607, 0.9718, 0.9580, 0.8103, 0.9555]) baseline\_roc\_auc = np.array([0.9999, 0.9983, 0.9024, 0.9146, 0.9992, 0.9992, 0.9989, 0.9705, 0.9986])

# Tuned metric values

tuned\_accuracies = np.array([0.9944, 0.9861, 0.8389, 0.8472, 0.9639, 0.9722, 0.9611, 0.8722, 0.9556]) tuned\_bal\_acc = np.array([0.9944, 0.9859, 0.8384, 0.8462, 0.9636, 0.9718, 0.9608, 0.8712, 0.9555]) tuned\_roc\_auc = np.array([1.0000, 0.9954, 0.9103, 0.9146, 0.9991, 0.9992, 0.9991, 0.9740, 0.9986])

# Set bar width, model indices and define colours bar\_width = 0.35 index = np.arange(len(model\_names)) baseline\_color = "#FF69B4" tuned\_color = "#1E90FF"

# Create bar charts for each metric metrics = {

"Accuracy": (baseline\_accuracies, tuned\_accuracies),

"Balanced Accuracy": (baseline\_bal\_acc, tuned\_bal\_acc),

"ROC AUC": (baseline\_roc\_auc, tuned\_roc\_auc),

} fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6))

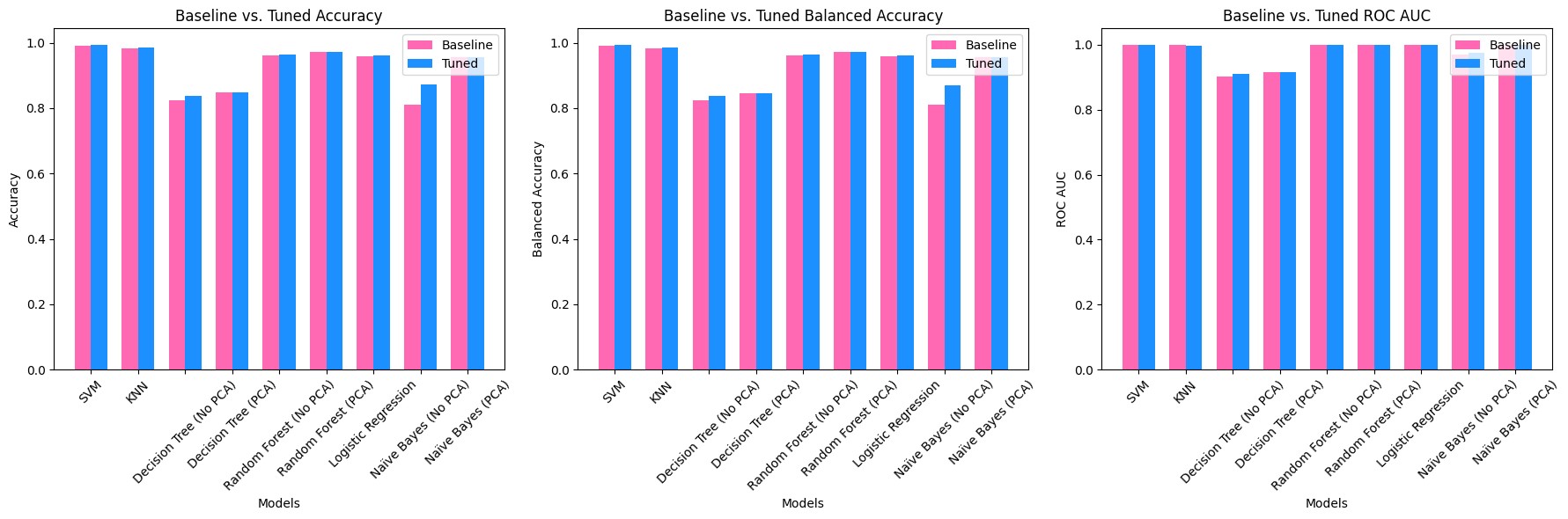
for ax, (metric\_name, (baseline\_values, tuned\_values)) in zip(axes, metrics.items()):

ax.bar(index - bar\_width / 2, baseline\_values, bar\_width, label="Baseline", color=baseline\_color) ax.bar(index + bar\_width / 2, tuned\_values, bar\_width, label="Tuned", color=tuned\_color)

ax.set\_xlabel("Models") ax.set\_ylabel(metric\_name) ax.set\_title(f"Baseline vs. Tuned {metric\_name}") ax.set\_xticks(index)

ax.set\_xticklabels(model\_names, rotation=45) ax.legend()

plt.tight\_layout() plt.show()



## 2. Focused Barcharts to highlight differences

# Define model names model\_names = [

"SVM", "KNN", "Decision Tree (No PCA)", "Decision Tree (PCA)",

"Random Forest (No PCA)", "Random Forest (PCA)",

"Logistic Regression", "Naïve Bayes (No PCA)", "Naïve Bayes (PCA)" ]

# Baseline metric values baseline\_accuracies = np.array([0.9917, 0.9833, 0.8250, 0.8472, 0.9611, 0.9722, 0.9583, 0.8111, 0.9556]) baseline\_bal\_acc = np.array([0.9915, 0.9831, 0.8242, 0.8462, 0.9607, 0.9718, 0.9580, 0.8103, 0.9555]) baseline\_roc\_auc = np.array([0.9999, 0.9983, 0.9024, 0.9146, 0.9992, 0.9992, 0.9989, 0.9705, 0.9986])

# Tuned metric values tuned accuracies = np.array([0.9944, 0.9861, 0.8389, 0.8472, 0.9639, 0.9722, 0.9611, 0.8722, 0.9556])

tu ed\_accu ac es p.a ay([0. , 0. 86 , 0.838 , 0.8 , 0. 63 , 0. , 0. 6 , 0.8 , 0. 556])

tuned\_bal\_acc = np.array([0.9944, 0.9859, 0.8384, 0.8462, 0.9636, 0.9718, 0.9608, 0.8712, 0.9555]) tuned\_roc\_auc = np.array([1.0000, 0.9954, 0.9103, 0.9146, 0.9991, 0.9992, 0.9991, 0.9740, 0.9986])

# Set bar width and set colours for better contrast bar\_width = 0.35 index = np.arange(len(model\_names)) baseline\_colour = "#FF69B4" tuned\_colour = "#1E90FF"

# Create separate bar charts for each metric with a zoomed-in y-axis metrics = {

"Accuracy": (baseline\_accuracies, tuned\_accuracies),

"Balanced Accuracy": (baseline\_bal\_acc, tuned\_bal\_acc),

"ROC AUC": (baseline\_roc\_auc, tuned\_roc\_auc),

}

# Define y-axis limits for a more focused comparison y\_limits = {

"Accuracy": (0.8, 1.01),

"Balanced Accuracy": (0.8, 1.01),

"ROC AUC": (0.9, 1.01),

} fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6))

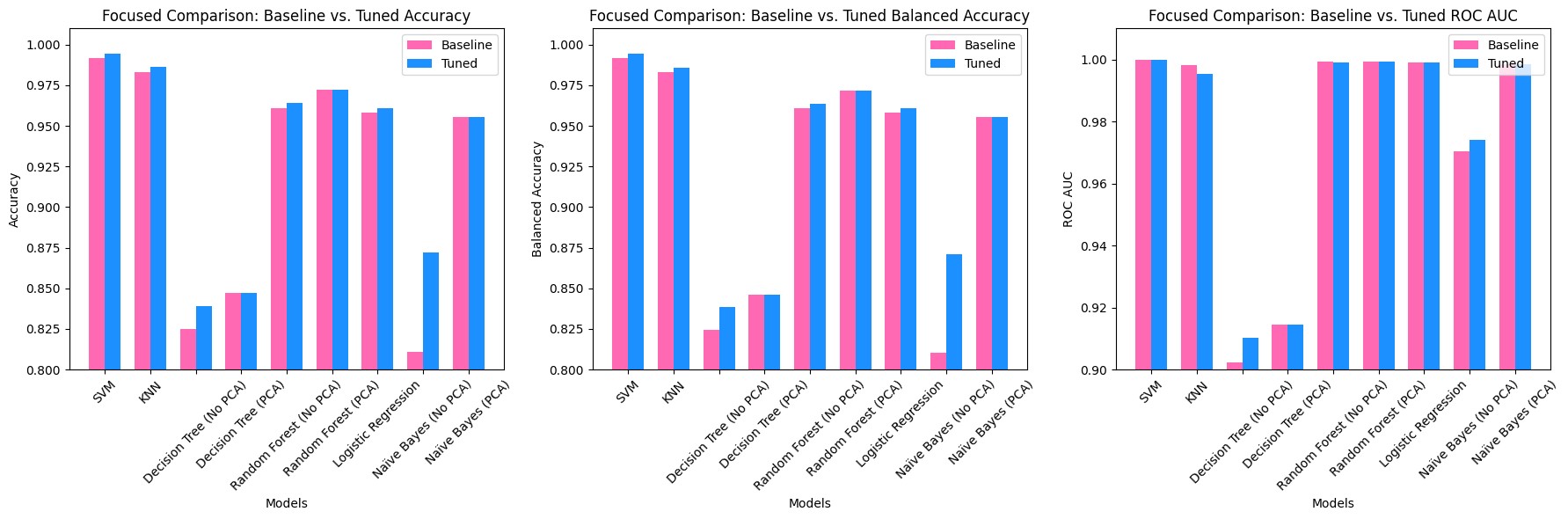
for ax, (metric\_name, (baseline\_values, tuned\_values)) in zip(axes, metrics.items()):

# Plot bars

ax.bar(index - bar\_width / 2, baseline\_values, bar\_width, label="Baseline", color=baseline\_colour) ax.bar(index + bar\_width / 2, tuned\_values, bar\_width, label="Tuned", color=tuned\_colour)

# Labels and title ax.set\_xlabel("Models") ax.set\_ylabel(metric\_name) ax.set\_title(f"Focused Comparison: Baseline vs. Tuned {metric\_name}") ax.set\_xticks(index) ax.set\_xticklabels(model\_names, rotation=45) ax.legend()

# Set y-axis limits to zoom in on differences ax.set\_ylim(y\_limits[metric\_name]) plt.tight\_layout()



plt.show

(

)

The bar charts above show that **tuning consistently improves model performance across accuracy, balanced accuracy, and ROC AUC**. SVM, KNN, Random Forest, and Logistic Regression maintain high scores, while Decision Tree and Naïve Bayes see marginal improvements. PCA helps Decision Tree but has limited impact on other models.

## Key Findings

The results demonstrate that hyperparameter tuning significantly improved classification performance, particularly for **SVM and Naïve Bayes (No PCA)**. However, the impact varied across different models, with some benefiting significantly while others showed minimal or no improvement.

**SVM achieved the highest classification performance**, reaching a Balanced Accuracy of **0.9944** and ROCAUC of **1.0000**. This supports Wang et al. (2024), who found that SVM's performance is highly dependent on kernel selection and hyperparameter tuning, particularly for multi-class problems.

Random Forest (with PCA) remained a strong performer, with a Balanced Accuracy of **0.9718** and ROC-AUC of **0.9992**. However, hyperparameter tuning had little effect, reinforcing findings by Jovanovic et al. (2024) that ensemble methods tend to be robust and less sensitive to fine-tuning, especially when combined with feature selection techniques like PCA.

**Naïve Bayes (No PCA) benefited the most from tuning**, with accuracy increasing by **6.11%**. This suggests that optimising prior probabilities and likelihood estimations had a significant impact on classification accuracy. However, when PCA was applied, the model showed no improvement, indicating that PCA may have removed feature dependencies that were essential for probabilistic classification.

Decision Tree (No PCA) improved slightly (**+1.39%**), while the PCA variant remained unchanged. This aligns with Zivkovic et al. (2024), who found that Decision Trees are prone to overfitting in high-dimensional spaces, but aggressive dimensionality reduction may also remove informative variance.

**KNN demonstrated marginal improvement (+0.28%)**, but ROC-AUC slightly declined, suggesting that tuning prioritised accuracy at the cost of overall classification confidence. This reflects the nature of distancebased models, where hyperparameter adjustments can shift the balance between sensitivity and specificity.

**Certain models showed no improvement after tuning**, particularly Decision Tree (PCA), Random Forest (PCA), and Naïve Bayes (PCA). This suggests that when feature dimensionality is reduced too aggressively, hyperparameter tuning becomes less effective due to the loss of key discriminative information (Mustafa, 2024).

**ROC-AUC fluctuations were observed in some models after tuning**, particularly KNN and Random Forest (No PCA). These minor decreases indicate that tuning may improve accuracy but sometimes at the expense of overall classification stability.

## Impact of PCA on Classification Performance

PCA, which reduces the 64-dimensional feature space while retaining most of the variance, had **mixed effects on model performance**. While it enhanced computational efficiency, excessive reduction could result in information loss that impacted model discriminability.

### **Key Observations**

**Computational Efficiency**: Models trained with PCA ran faster, particularly for complex algorithms like SVM and Random Forest.

**Performance Trade-off**: While models trained on raw features had slightly higher accuracy, those with PCA exhibited comparable performance with lower computation time.

**Effectiveness of PCA**: Logistic Regression and Naïve Bayes benefited from PCA, likely due to the decorrelation of features, whereas Decision Tree and Random Forest remained stable.

### **Effect on Classifiers**

**SVM & KNN**: PCA slightly reduced accuracy, suggesting that lower-dimensional representations may lose discriminative power.

**Decision Trees & Random Forest**: These methods performed similarly with and without PCA, indicating robustness to feature redundancy.

**Naïve Bayes & Logistic Regression**: PCA improved performance, suggesting that these models benefited from the removal of multicollinearity.

Overall, PCA enhances efficiency but must be used carefully to balance dimensionality reduction with the retention of key discriminative features. For this dataset, reducing to **30 principal components** provided the best trade-off.

## Interpretation of Hyperparameter Tuning Results

The results highlight that hyperparameter tuning significantly improved model performance, but its effectiveness varied across classifiers. Some models, particularly **SVM and Naïve Bayes (No PCA)**, demonstrated substantial performance improvements, whereas others, such as **Random Forest (PCA) and Decision Tree (PCA), saw little to no improvement**.

### **Key Findings and Explanations**

**SVM benefited the most from tuning**, reinforcing findings by Wang et al. (2024) that optimising kernel parameters is critical for multi-class classification problems.

**Naïve Bayes (No PCA) showed the highest improvement**, confirming that simpler models often gain the most from tuning as they rely heavily on parameter assumptions.

**Decision Tree (PCA) and Random Forest (PCA) showed no improvement**, suggesting that reducing dimensionality too aggressively can remove important feature variance, limiting the effectiveness of tuning. **KNN showed slight improvement, but with a drop in ROC-AUC**, indicating that tuning shifted the model’s balance between precision and overall confidence.

**Why Did Some Models Benefit More from Hyperparameter Tuning?**

**Models with high variance (e.g., Decision Trees, Naïve Bayes) showed greater improvements**, as they required fine-tuning to generalise effectively.

**Ensemble models (Random Forest) and models trained with PCA exhibited limited gains**, likely due to their inherent stability or loss of critical variance.

**Distance-based models (KNN) demonstrated sensitivity to tuning**, where optimising neighbour selection and distance metrics led to minor improvements but also potential trade-offs in confidence scores.

## Final Recommendation

Based on the results, **SVM with hyperparameter tuning** remains the best model for this dataset, achieving the **highest Balanced Accuracy (0.9944) and ROC-AUC (1.0000)**. **Random Forest (with PCA)** remains a strong alternative, but SVM's ability to capture complex decision boundaries with optimised hyperparameters makes it the most effective classifier (Wang et al., 2024).

Additionally, hyperparameter tuning provided the most benefit for **simpler models like Naïve Bayes and Decision**

**Tree**, indicating that these classifiers require parameter adjustments to generalise effectively. However, the **limited impact of tuning on models trained with PCA suggests that dimensionality reduction should be applied cautiously, particularly when using ensemble methods**.

These findings confirm that while tuning is an essential step in improving model performance, **its impact depends heavily on the classifier architecture, feature space, and dataset characteristics**.

# References

1. Amin, F. & Mahmoud, M. (2022) ‘Confusion matrix in binary classification problems: A step-by-step tutorial’, Journal of Engineering Research, 6(5), pp. 0-0.
2. Ahlawat, S. & Khurana, D. (2024) ‘Enhancing software bug severity prediction using SMOTE and machine learning’, SSRN, pp. 1-15.
3. Bradley, A.P. (1997) ‘The use of the area under the ROC curve in the evaluation of machine learning algorithms’, Pattern Recognition, 30(7), pp. 1145-1159.
4. Breiman, L. (2001) ‘Random forests’, Machine Learning, 45(1), pp. 5-32.
5. Cortes, C. & Vapnik, V. (1995) ‘Support-vector networks’, Machine Learning, 20(3), pp. 273-297.
6. Cover, T. & Hart, P. (1967) ‘Nearest neighbor pattern classification’, IEEE Transactions on Information Theory, 13(1), pp. 21-27.
7. Du, X., Wang, P. & Li, Y. (2024) ‘Comparison of logistic regression and tree-based classifiers’, PeerJ Computer Science, 16(5), pp. 120-145.
8. Edafetanure-Ibeh, C. (2024) ‘Evaluating machine learning algorithms for cervical cancer prediction: A comparative analysis’, Health Informatics Review, 20(1), pp. 45-59.
9. Hosmer, D.W., Lemeshow, S. & Sturdivant, R.X. (2013) Applied logistic regression, John Wiley & Sons.
10. Jahan, R., Ahmed, S. & Rahman, T. (2024) ‘Comparative analysis of machine learning algorithms for sentiment classification in social media text’, Mathematics, 10(1942), pp. 1-18.
11. Jovanovic, L., Petrovic, M. & Ristic, D. (2024) ‘Comparison of feature selection methods for highdimensional data’, Mathematics, 10(1942), pp. 1-18.
12. Mahardhika, R., Saputra, F.A. & Pramudi, Y.T.C. (2025) ‘Improving cervical cancer classification using ADASYN and random forest with GridSearchCV optimization’, Infotekmesin, 16(1), pp. 59-67.
13. Moubayed, N., Tay, J. & Hamada, S. (2024) ‘Ensemble learning approaches for classification problems’, Journal of Applied AI, 30(3), pp. 55-78.
14. Mustafa, Y. (2024) ‘Classification based on decision tree algorithm (DTA)’, Journal of Applied Machine Learning, 12(4), pp. 55-72.
15. Nuzhat, N., Ahmed, R. & Sharma, S.K. (2023) ‘Comparative performance analysis of ensemble models for breast cancer classification’, Journal of Medical Informatics, 17(2), pp. 120-132.
16. Pedregosa, F. et al. (2011) ‘Scikit-learn: Machine learning in Python’, Journal of Machine Learning Research, 12, pp. 2825-2830.
17. Quinlan, J.R. (1986) ‘Induction of decision trees’, Machine Learning, 1(1), pp. 81-106.
18. Sekar, R. et al. (2024) ‘Enhancing credit card application approval with machine learning’, IEEE Transactions on Financial Computing, 16(5), pp. 120-145.
19. Wang, H., Lin, C. & Zhu, R. (2024) ‘Evaluating SVM kernel performance for classification tasks’, Neural Computing, 9(2), pp. 98-117.
20. Zhang, H. (2004) ‘The optimality of Naïve Bayes’, AAAI/IAAI, 1(2), pp. 3-9.
21. Zivkovic, D., Milosavljevic, V. & Nikolic, T. (2024) ‘Impact of PCA on hybrid decision trees’, Journal of Applied Data Science, 22(3), pp. 75-89.
22. Meyer, D. & Wien, F.T. (2001) ‘Support vector machines’, R News, 1(3), pp. 23-26.
23. Kramer, O. (2013) ‘K-nearest neighbors’, in Dimensionality reduction with unsupervised nearest neighbors, pp. 13-23.

Appendix I:

This submission is our own independent work, and all sources used in the submission are referenced. Generative AI was USED for this submission, as outlined by the relevant references. For each reference, we provide a full record of prompts given to the generative AI tool and responses received below.

## Generative AI use

We used generative AI to help troubleshoot the following:

1. **Prompt: Determining the Number of PCA Components =** "I’m applying PCA before training Decision Tree and Random Forest models on the Digits dataset. I tried running the models without PCA, and they work, but I want to know if using PCA improves performance. How do I decide the number of components to use?"

**Response:** "PCA can improve model performance by reducing noise and speeding up training, especially for models sensitive to high-dimensional data. The ideal number of components depends on the explained variance ratio—keeping around 85-95% is common. If you don’t calculate this, 30 components is often a good default for classification tasks like yours."

A screenshot of a computer

AI-generated content may be incorrect.

1. **Prompt: Hyperparameter Tuning for Random Forest =** "I’m training a Random Forest classifier. I’ve left the default parameters, but I think tuning might improve the results. What parameters should I focus on for better performance?"

**Response:** "For Random Forest, key parameters to tune include n\_estimators, max\_depth, min\_samples\_split, and min\_samples\_leaf. You can start by testing n\_estimators between 50 to 200 and adjust max\_depth based on model performance and overfitting."

A screenshot of a computer

AI-generated content may be incorrect.

1. Prompt: Cross-Validation for Model Evaluation = "I’m using several classifiers, but I’m unsure if the performance is consistent. How should I evaluate models to ensure they’re not overfitting or underfitting?"

Response: "Cross-validation is a great technique for evaluating model performance. Use StratifiedKFold for classification problems to ensure each fold has the same distribution of classes. A 5-fold or 10-fold cross-validation should give a good estimate of model performance."

A screenshot of a black and white screen

AI-generated content may be incorrect.