Sri Sivasubramaniya Nadar College of Engineering, Chennai

(An autonomous Institution affiliated to Anna University)

Degree & Branch: B.E. Computer Science & Engineering
Semester: V
Subject Code & Name: ICS1512 - Machine Learning Algorithms Laboratory
Academic Year: 2025–2026 (Odd)
Batch: 2023–2028

Experiment 6: Dimensionality Reduction and Model Evaluation (With and Without PCA)

Aim

To study the effect of dimensionality reduction using Principal Component Analysis (PCA) on the performance of various machine learning classifiers. The task requires:

1. Training and validating models without PCA (original feature space).
2. Training and validating models with PCA (reduced feature space).

For both cases, students must perform hyperparameter tuning, apply 5-fold cross-validation, and record performance.

Libraries Used

- Pandas
- NumPy
- Matplotlib
- Scikit-learn
- XGBoost

Objective

To evaluate how dimensionality reduction using Principal Component Analysis (PCA) influences the accuracy and generalization of different machine learning classifiers, by comparing their performance with and without PCA through hyperparameter tuning and 5-fold cross-validation. [a4paper,12pt]article pdfpages

Including PDF

Here is my PDF included below:

Experiment 6: Dimensionality Reduction and Model Evaluation (With and Without PCA)

```
# IMPORTS
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV, cross_va
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import f1_score, accuracy_score, make_scorer, roc_curve,
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier, R
from xgboost import XGBClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
```

```
TARGET_COLUMN = 'class'

df = pd.read_csv('/content/drive/MyDrive/ml-lab/spambase_csv.csv')
print(df.head())
print("Initial shape:", df.shape)
```

```
   word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
0            0.00               0.64           0.64           0.0
1            0.21               0.28           0.50           0.0
2            0.06               0.00           0.71           0.0
3            0.00               0.00           0.00           0.0
4            0.00               0.00           0.00           0.0
```

```
   word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
0           0.32            0.00              0.00                0.00
1           0.14            0.28              0.21                0.07
2           1.23            0.19              0.19                0.12
3           0.63            0.00              0.31                0.63
4           0.63            0.00              0.31                0.63

   word_freq_order  word_freq_mail  ...  char_freq_%3B  char_freq_%28  \
0             0.00            0.00  ...           0.00          0.000
1             0.00            0.94  ...           0.00          0.132
2             0.64            0.25  ...           0.01          0.143
3             0.31            0.63  ...           0.00          0.137
4             0.31            0.63  ...           0.00          0.135

   char_freq_%5B  char_freq_%21  char_freq_%24  char_freq_%23  \
0            0.0          0.778          0.000          0.000
1            0.0          0.372          0.180          0.048
2            0.0          0.276          0.184          0.010
3            0.0          0.137          0.000          0.000
4            0.0          0.135          0.000          0.000

   capital_run_length_average  capital_run_length_longest  \
0                       3.756                          61
1                       5.114                         101
2                       9.821                         485
3                       3.537                          40
4                       3.537                          40

   capital_run_length_total  class
0                       278      1
1                      1028      1
2                      2259      1
3                       191      1
4                       191      1

[5 rows x 58 columns]
Initial shape: (4601, 58)
```

```
# HANDLE MISSING VALUES
df = df.dropna(thresh=df.shape[1]//2)  # Drop rows with >50% missing
df.fillna(df.median(numeric_only=True), inplace=True)
```

```
# OUTLIER HANDLING (Z-Score)
def remove_outliers(df, threshold=3):
    numeric_cols = df.select_dtypes(include=[np.number]).columns
    z_scores = np.abs((df[numeric_cols] - df[numeric_cols].mean()) / df[numeric_cols].std())
    return df[(z_scores < threshold).all(axis=1)]

df = remove_outliers(df)
# print("After outlier removal:", df.shape)
```

```
# FEATURE / TARGET SPLIT
X = df.drop(columns=[TARGET_COLUMN])
y = df[TARGET_COLUMN]

# ENCODE + STANDARDIZE
numeric_cols = X.select_dtypes(include=[np.number]).columns
categorical_cols = X.select_dtypes(exclude=[np.number]).columns
X_encoded = pd.get_dummies(X, columns=categorical_cols)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_encoded)

# TRAIN / TEST SPLIT
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled,
    y,
    test_size=0.2,          # 20% test hold-out
    stratify=y,             # keep class balance if classification
    random_state=42
)
```

PCA Variance Explained

```
pca = PCA(n_components=0.95)
pca.fit(X_scaled)

print("Chosen components:", pca.n_components_)
print("Total variance explained (%):", pca.explained_variance_ratio_.sum()*100)
```

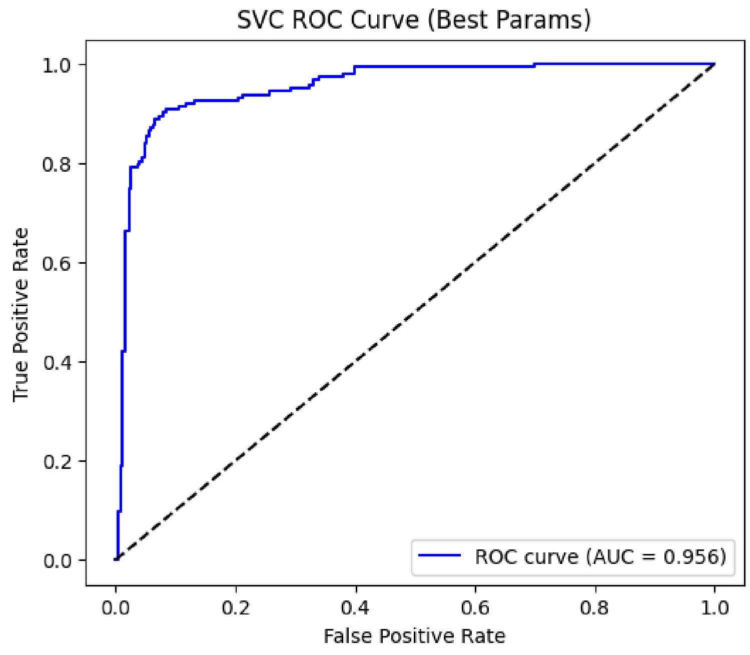
```
Chosen components: 49
Total variance explained (%): 95.53617010131482
```

Support Vector Machine (SVM)

```
# ========== HYPERPARAM GRID ==========
param_grid = {
    'kernel': ['linear', 'rbf'],
    'C': [0.1, 10],
    'gamma': ['scale']
}

# ========== HELPER FUNCTION ==========
def evaluate_svc(X_train, X_test, y_train, y_test, use_pca=False, pca_variance=0.95):
    if use_pca:
        pca = PCA(n_components=pca_variance)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc = X_train
        X_test_proc = X_test

    svc = SVC(probability=True)
    grid = GridSearchCV(svc, param_grid, cv=5, scoring='accuracy')  # you can switch scoring
    grid.fit(X_train_proc, y_train)

    best_model = grid.best_estimator_
    y_pred = best_model.predict(X_test_proc)
    y_proba = best_model.predict_proba(X_test_proc)[:,1]

    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba)

    return grid.best_params_, acc, auc, y_test, y_proba

# ========== EVALUATE ALL COMBOS ==========
results = []
for kernel in param_grid['kernel']:
    for C in param_grid['C']:
        for gamma in param_grid['gamma']:
            params = {'kernel': kernel, 'C': C, 'gamma': gamma}

            # No PCA
            _, acc_no_pca, auc_no_pca, y_test_val, y_proba_no_pca = evaluate_svc(
                X_train, X_test, y_train, y_test, use_pca=False
            )

            # With PCA
            _, acc_pca, auc_pca, _, y_proba_pca = evaluate_svc(
                X_train, X_test, y_train, y_test, use_pca=True, pca_variance=0.95
            )

            results.append({
                'kernel': kernel,
                'C': C,
                'gamma': gamma,
                'Accuracy_no_PCA': acc_no_pca,
                'Accuracy_PCA': acc_pca
            })

# ========== RESULTS TABLE ==========
results_df = pd.DataFrame(results)
print("SVC Performance Table")
print(results_df)

# ========== BEST MODEL ==========
best_idx = results_df['Accuracy_no_PCA'].idxmax()  # you can also pick max of PCA
best_params = results_df.iloc[best_idx]
print("\nBest Params (No PCA)")
print(best_params)

# ========== ROC CURVE FOR BEST MODEL ==========
# Using No PCA best model
best_kernel = best_params['kernel']
best_C = best_params['C']
best_gamma = best_params['gamma']
```

```
svc_best = SVC(kernel=best_kernel, C=best_C, gamma=best_gamma, probability=True)
svc_best.fit(X_train, y_train)
y_proba_best = svc_best.predict_proba(X_test)[:,1]
fpr, tpr, _ = roc_curve(y_test, y_proba_best)
roc_auc = roc_auc_score(y_test, y_proba_best)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.3f})', color='blue')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('SVC ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

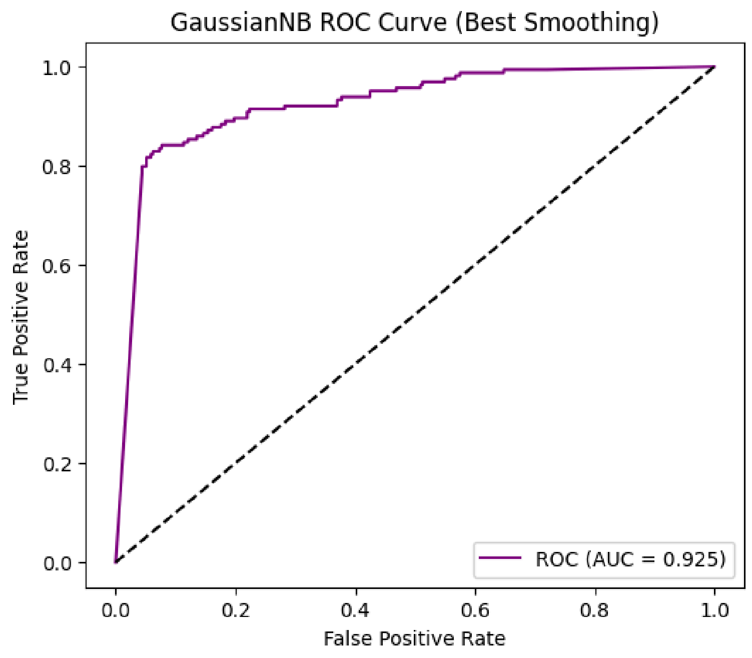
```
SVC Performance Table
   kernel     C  gamma  Accuracy_no_PCA  Accuracy_PCA
0  linear   0.1  scale          0.93135      0.924485
1  linear  10.0  scale          0.93135      0.924485
2     rbf   0.1  scale          0.93135      0.924485
3     rbf  10.0  scale          0.93135      0.924485

Best Params (No PCA)
kernel               linear
C                       0.1
gamma                 scale
Accuracy_no_PCA     0.93135
Accuracy_PCA       0.924485
Name: 0, dtype: object
```



Naive Bayes

```
# Smoothing values to test
smoothing_values = [1e-9, 1e-8, 1e-7, 1e-6]

def evaluate_nb(X_train, X_test, y_train, y_test, smoothing, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc = X_train
        X_test_proc = X_test

    model = GaussianNB(var_smoothing=smoothing)
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_prob)
    return acc, auc, y_prob

results = []
for s in smoothing_values:
    acc_no_pca, auc_no_pca, prob_no_pca = evaluate_nb(
        X_train, X_test, y_train, y_test, s, use_pca=False
    )
    acc_pca, auc_pca, prob_pca = evaluate_nb(
        X_train, X_test, y_train, y_test, s, use_pca=True
    )
    results.append({
        "smoothing": s,
        "Accuracy_no_PCA": acc_no_pca,
        "Accuracy_PCA": acc_pca
    })

results_df = pd.DataFrame(results)
print("Naive Bayes Performance Table")
print(results_df)

# --- Best Params (based on no PCA accuracy) ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_params = results_df.iloc[best_idx]
print("\nBest Smoothing (No PCA)")
print(best_params)

# --- ROC curve for best model ---
best_s = best_params['smoothing']
nb_best = GaussianNB(var_smoothing=best_s)
nb_best.fit(X_train, y_train)
y_proba_best = nb_best.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba_best)
roc_auc = roc_auc_score(y_test, y_proba_best)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='purple')
plt.plot([0,1],[0,1],'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('GaussianNB ROC Curve (Best Smoothing)')
plt.legend(loc='lower right')
plt.show()
```

```
Naive Bayes Performance Table
      smoothing  Accuracy_no_PCA  Accuracy_PCA
0  1.000000e-09         0.720824      0.832952
1  1.000000e-08         0.723112      0.832952
2  1.000000e-07         0.725400      0.832952
3  1.000000e-06         0.725400      0.832952

Best Smoothing (No PCA)
smoothing          1.000000e-07
Accuracy_no_PCA    7.254005e-01
Accuracy_PCA       8.329519e-01
Name: 2, dtype: float64
```



K-Nearest Neighbours

```
# --- Hyperparam grid ---
k_values = [3, 5]
weights_options = ['uniform', 'distance']
metrics_options = ['euclidean', 'manhattan']

def evaluate_knn(X_train, X_test, y_train, y_test,
                 k, weight, metric, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    model = KNeighborsClassifier(n_neighbors=k,
                                 weights=weight,
                                 metric=metric)
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_prob)
    return acc, auc, y_prob

results = []
for k in k_values:
    for w in weights_options:
        for m in metrics_options:
            acc_no_pca, auc_no_pca, _ = evaluate_knn(
                X_train, X_test, y_train, y_test,
                k, w, m, use_pca=False
            )
            acc_pca, auc_pca, _ = evaluate_knn(
                X_train, X_test, y_train, y_test,
                k, w, m, use_pca=True
            )
            results.append({
                "k": k,
                "weights": w,
                "metric": m,
                "Accuracy_no_PCA": acc_no_pca,
                "Accuracy_PCA": acc_pca
            })

results_df = pd.DataFrame(results)
print("KNN Performance Table")
print(results_df)

# --- Pick best based on no-PCA accuracy ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_params = results_df.iloc[best_idx]
print("\nBest KNN Params (No PCA)")
print(best_params)

# --- ROC curve for that best combo ---
best_k = best_params['k']
best_w = best_params['weights']
best_m = best_params['metric']

knn_best = KNeighborsClassifier(n_neighbors=best_k,
                                weights=best_w,
                                metric=best_m)
knn_best.fit(X_train, y_train)
y_proba_best = knn_best.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba_best)
roc_auc = roc_auc_score(y_test, y_proba_best)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='green')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('KNN ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

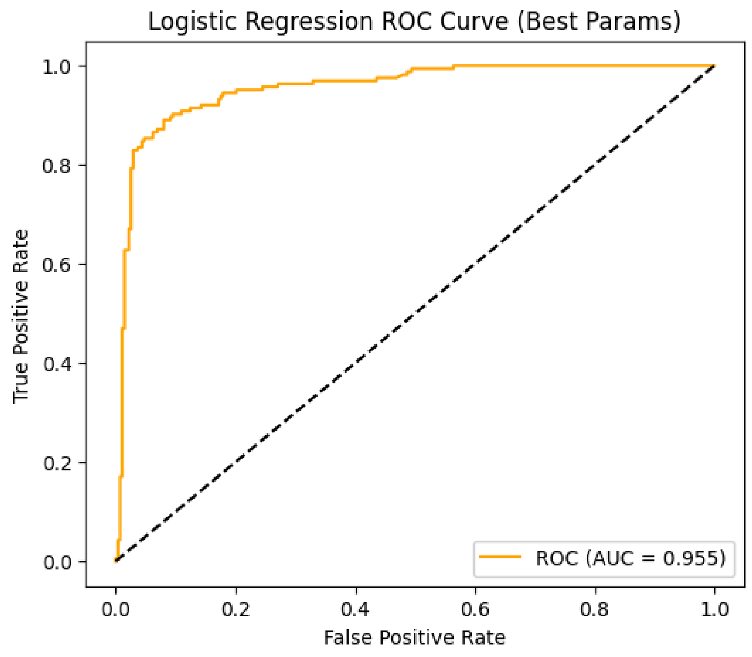
```
KNN Performance Table
   k   weights     metric  Accuracy_no_PCA  Accuracy_PCA
0  3   uniform  euclidean         0.899314      0.901602
1  3   uniform  manhattan         0.899314      0.897025
2  3  distance  euclidean         0.910755      0.908467
```

Logistic Regression

```
# --- Hyperparam grid ---
c_values = [0.01, 0.1, 1]
penalties = ['l2', 'l1']

def evaluate_logreg(X_train, X_test, y_train, y_test,
                    c_val, penalty, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    solver = 'saga' if penalty == 'l1' else 'lbfgs'
    model = LogisticRegression(C=c_val,
                               penalty=penalty,
                               solver=solver,
                               max_iter=5000)
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_prob)
    return acc, auc, y_prob

results = []
for c in c_values:
    for p in penalties:
        acc_no_pca, auc_no_pca, _ = evaluate_logreg(
            X_train, X_test, y_train, y_test,
            c, p, use_pca=False
        )
        acc_pca, auc_pca, _ = evaluate_logreg(
            X_train, X_test, y_train, y_test,
            c, p, use_pca=True
        )
        results.append({
            "C": c,
            "penalty": p,
            "Accuracy_no_PCA": acc_no_pca,
            "Accuracy_PCA": acc_pca
        })

results_df = pd.DataFrame(results)
print("Logistic Regression Performance Table")
print(results_df)

# --- Pick best by no-PCA accuracy ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_params = results_df.iloc[best_idx]
print("\nBest Logistic Regression Params (No PCA)")
print(best_params)

# --- ROC curve for best combo ---
best_c = best_params['C']
best_p = best_params['penalty']
solver = 'saga' if best_p == 'l1' else 'lbfgs'

log_best = LogisticRegression(C=best_c,
                              penalty=best_p,
                              solver=solver,
                              max_iter=5000)
log_best.fit(X_train, y_train)
y_proba_best = log_best.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba_best)
roc_auc = roc_auc_score(y_test, y_proba_best)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='orange')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression ROC Curve (Best Params)')
```

```
    plt.legend(loc='lower right')
    plt.show()
```

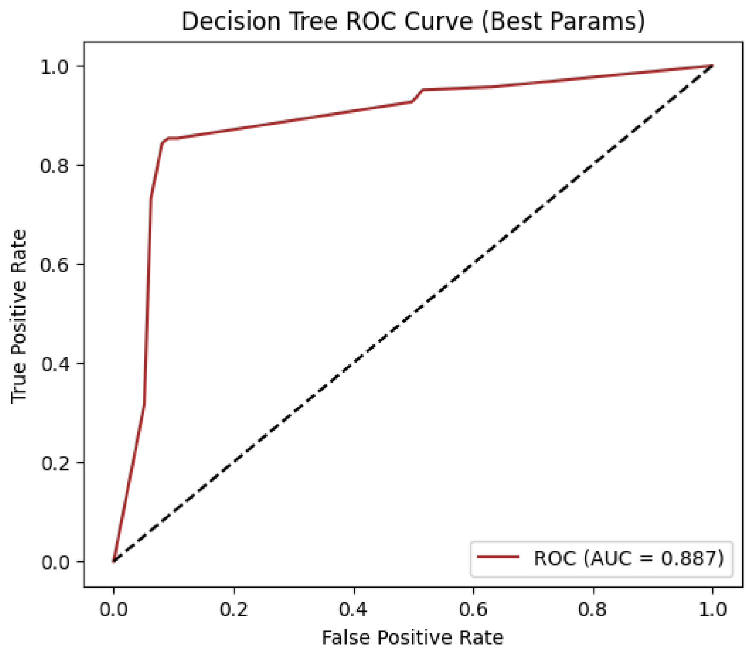
```
Logistic Regression Performance Table
      C penalty  Accuracy_no_PCA  Accuracy_PCA
0  0.01      l2         0.908467      0.908467
1  0.01      l1         0.871854      0.894737
2  0.10      l2         0.903890      0.906178
3  0.10      l1         0.913043      0.917620
4  1.00      l2         0.910755      0.901602
5  1.00      l1         0.910755      0.906178

Best Logistic Regression Params (No PCA)
C                        0.1
penalty                   l1
Accuracy_no_PCA     0.913043
Accuracy_PCA         0.91762
Name: 3, dtype: object
```



Decision Tree

```
# --- Hyperparams to explore ---
criteria = ["gini", "entropy"]
depths = [None, 5, 10]

def evaluate_dt(X_train, X_test, y_train, y_test,
                criterion, depth, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    model = DecisionTreeClassifier(
        criterion=criterion,
        max_depth=depth,
        random_state=42
    )
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_prob)
    return acc, auc, y_prob

results = []
for c in criteria:
    for d in depths:
        acc_no_pca, auc_no_pca, _ = evaluate_dt(
            X_train, X_test, y_train, y_test,
            c, d, use_pca=False
        )
        acc_pca, auc_pca, _ = evaluate_dt(
            X_train, X_test, y_train, y_test,
            c, d, use_pca=True
        )
        results.append({
            "criterion": c,
            "max_depth": d,
            "Accuracy_no_PCA": acc_no_pca,
            "Accuracy_PCA": acc_pca
        })

results_df = pd.DataFrame(results)
print("Decision Tree Performance Table")
print(results_df)

# --- Best by no-PCA accuracy ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_params = results_df.iloc[best_idx]
print("\nBest Decision Tree Params (No PCA)")
print(best_params)

# --- ROC curve for best combo ---
best_c = best_params['criterion']
best_d = best_params['max_depth']

# convert NaN to None, otherwise to int
if pd.isna(best_d):
    best_d = None
else:
    best_d = int(best_d)

dt_best = DecisionTreeClassifier(
    criterion=best_params['criterion'],
    max_depth=best_d,
    random_state=42
)
dt_best.fit(X_train, y_train)
y_proba_best = dt_best.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba_best)
roc_auc = roc_auc_score(y_test, y_proba_best)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='brown')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Decision Tree ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

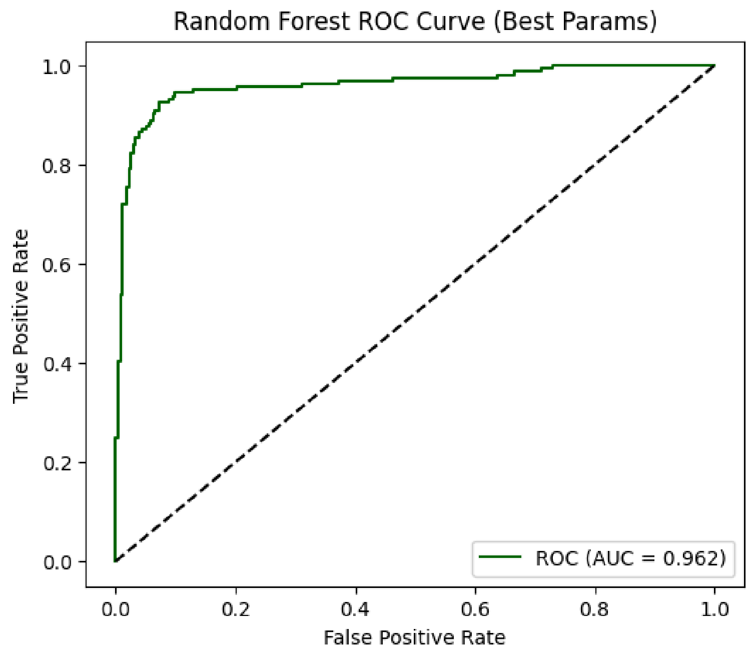
```
Decision Tree Performance Table
  criterion  max_depth  Accuracy_no_PCA  Accuracy_PCA
0      gini        NaN         0.887872      0.883295
1      gini        5.0         0.876430      0.860412
2      gini       10.0         0.890160      0.874142
3   entropy        NaN         0.855835      0.883295
4   entropy        5.0         0.871854      0.881007
5   entropy       10.0         0.878719      0.878719

Best Decision Tree Params (No PCA)
criterion              gini
max_depth              10.0
Accuracy_no_PCA     0.89016
Accuracy_PCA       0.874142
Name: 2, dtype: object
```



Random Forest

```
# --- Hyperparams to explore ---
n_estimators_list = [50, 100]
max_depth_list = [None, 5, 10]

def evaluate_rf(X_train, X_test, y_train, y_test,
                n_estimators, depth, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=depth,
        random_state=42,
        n_jobs=-1
    )
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_prob)
    return acc, auc, y_prob

results = []
for n in n_estimators_list:
    for d in max_depth_list:
        acc_no_pca, auc_no_pca, _ = evaluate_rf(
            X_train, X_test, y_train, y_test,
            n, d, use_pca=False
        )
        acc_pca, auc_pca, _ = evaluate_rf(
            X_train, X_test, y_train, y_test,
            n, d, use_pca=True
        )
        results.append({
            "n_estimators": n,
            "max_depth": d,
            "Accuracy_no_PCA": acc_no_pca,
            "Accuracy_PCA": acc_pca
        })

results_df = pd.DataFrame(results)
print("Random Forest Performance Table")
print(results_df)

# --- Best by no-PCA accuracy ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_params = results_df.iloc[best_idx]
print("\nBest Random Forest Params (No PCA)")
print(best_params)

# --- Fix dtype for max_depth ---
best_depth = best_params['max_depth']
if pd.isna(best_depth):
    best_depth = None
else:
    best_depth = int(best_depth)

# --- ROC curve for best combo ---
rf_best = RandomForestClassifier(
    n_estimators=int(best_params['n_estimators']),
    max_depth=best_depth,
    random_state=42,
    n_jobs=-1
)
rf_best.fit(X_train, y_train)
y_proba_best = rf_best.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_proba_best)
roc_auc = roc_auc_score(y_test, y_proba_best)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='darkgreen')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Random Forest ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

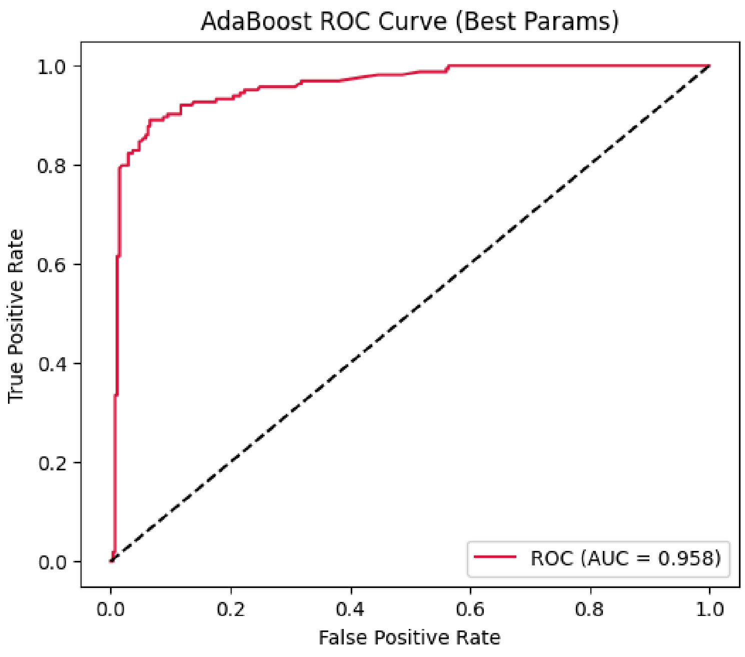
```
Random Forest Performance Table
   n_estimators  max_depth  Accuracy_no_PCA  Accuracy_PCA
0            50        NaN         0.922197      0.906178
1            50        5.0         0.910755      0.897025
2            50       10.0         0.924485      0.908467
3           100        NaN         0.924485      0.908467
4           100        5.0         0.908467      0.899314
5           100       10.0         0.922197      0.910755

Best Random Forest Params (No PCA)
n_estimators       50.000000
max_depth          10.000000
Accuracy_no_PCA     0.924485
Accuracy_PCA        0.908467
Name: 2, dtype: float64
```



AdaBoost

```
# --- Hyperparam grids ---
n_estimators_list = [50, 100]
learning_rates = [0.01, 0.1, 1.0]

def eval_adaboost(X_train, X_test, y_train, y_test,
                  n_est, lr, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    model = AdaBoostClassifier(
        n_estimators=n_est,
        learning_rate=lr,
        random_state=42
    )
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    return (
        accuracy_score(y_test, y_pred),
        roc_auc_score(y_test, y_prob)
    )

results = []
for n in n_estimators_list:
    for lr in learning_rates:
        acc_no_pca, _ = eval_adaboost(X_train, X_test, y_train, y_test,
                                      n, lr, use_pca=False)
        acc_pca, _ = eval_adaboost(X_train, X_test, y_train, y_test,
                                   n, lr, use_pca=True)
        results.append({
            "n_estimators": n,
            "learning_rate": lr,
            "Accuracy_no_PCA": acc_no_pca,
            "Accuracy_PCA": acc_pca
        })

results_df = pd.DataFrame(results)
print("AdaBoost Performance")
print(results_df)

# --- Best combo (no PCA) ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_row = results_df.iloc[best_idx]
print("\nBest Params (No PCA)")
print(best_row)

# --- ROC for the best ---
best_model = AdaBoostClassifier(
    n_estimators=int(best_row['n_estimators']),
    learning_rate=float(best_row['learning_rate']),
    random_state=42
)
best_model.fit(X_train, y_train)
y_best_prob = best_model.predict_proba(X_test)[:, 1]
fpr, tpr, _ = roc_curve(y_test, y_best_prob)
roc_auc = roc_auc_score(y_test, y_best_prob)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='crimson')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('AdaBoost ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

```
AdaBoost Performance
   n_estimators  learning_rate  Accuracy_no_PCA  Accuracy_PCA
0            50           0.01         0.821510      0.878719
1            50           0.10         0.897025      0.878719
2            50           1.00         0.908467      0.892449
3           100           0.01         0.844394      0.878719
4           100           0.10         0.899314      0.883295
5           100           1.00         0.915332      0.901602

Best Params (No PCA)
n_estimators       100.000000
learning_rate        1.000000
Accuracy_no_PCA      0.915332
Accuracy_PCA         0.901602
Name: 5, dtype: float64
```

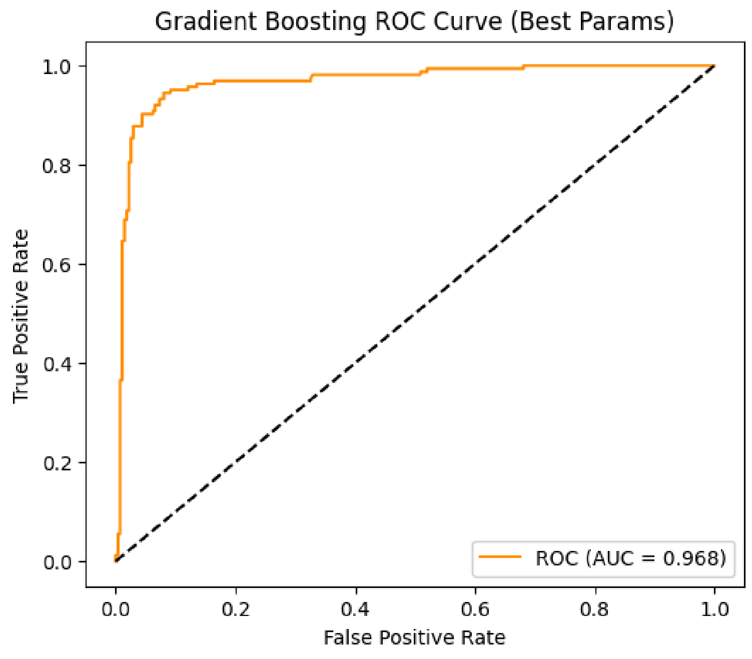


Gradient Boosting

```
# --- Hyperparam grids ---
n_estimators_list = [50, 100, 200]
learning_rates = [0.1, 0.2]

def eval_gb(X_train, X_test, y_train, y_test, n_est, lr, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    model = GradientBoostingClassifier(
        n_estimators=n_est,
        learning_rate=lr,
        random_state=42
    )
    model.fit(X_train_proc, y_train)
    y_pred = model.predict(X_test_proc)
    y_prob = model.predict_proba(X_test_proc)[:, 1]
    return accuracy_score(y_test, y_pred), roc_auc_score(y_test, y_prob)

# Collect results
results = []
for n in n_estimators_list:
    for lr in learning_rates:
        acc_no_pca, _ = eval_gb(X_train, X_test, y_train, y_test,
                                n, lr, use_pca=False)
        acc_pca, _ = eval_gb(X_train, X_test, y_train, y_test,
                             n, lr, use_pca=True)
        results.append({
            "n_estimators": n,
            "learning_rate": lr,
            "Accuracy_no_PCA": acc_no_pca,
            "Accuracy_PCA": acc_pca
        })
```

```
results_df = pd.DataFrame(results)
print("Gradient Boosting Performance")
print(results_df)

# --- Best combo (no PCA) ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_row = results_df.iloc[best_idx]
print("\nBest Params (No PCA)")
print(best_row)

# --- ROC curve for best model ---
best_model = GradientBoostingClassifier(
    n_estimators=int(best_row["n_estimators"]),
    learning_rate=float(best_row["learning_rate"]),
    random_state=42
)
best_model.fit(X_train, y_train)
y_best_prob = best_model.predict_proba(X_test)[:, 1]

fpr, tpr, _ = roc_curve(y_test, y_best_prob)
roc_auc = roc_auc_score(y_test, y_best_prob)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})', color='darkorange')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Gradient Boosting ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

```
Gradient Boosting Performance
   n_estimators  learning_rate  Accuracy_no_PCA  Accuracy_PCA
0            50            0.1         0.924485      0.901602
1            50            0.2         0.922197      0.913043
2           100            0.1         0.931350      0.908467
3           100            0.2         0.929062      0.913043
4           200            0.1         0.924485      0.908467
5           200            0.2         0.931350      0.910755

Best Params (No PCA)
n_estimators       100.000000
learning_rate        0.100000
Accuracy_no_PCA      0.931350
Accuracy_PCA         0.908467
Name: 2, dtype: float64
```



XGBoost

```
# --- Hyperparam grids ---
n_estimators_list = [50, 100]
learning_rates   = [0.1, 0.2]
max_depths       = [5, 7]

def eval_xgb(X_train, X_test, y_train, y_test,
             n_est, lr, depth, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        X_train_proc = pca.fit_transform(X_train)
        X_test_proc  = pca.transform(X_test)
    else:
        X_train_proc, X_test_proc = X_train, X_test

    model = XGBClassifier(
        n_estimators=n_est,
        learning_rate=lr,
        max_depth=depth,
        eval_metric='logloss',
        use_label_encoder=False,
        random_state=42
    )
    model.fit(X_train_proc, y_train)
    y_pred  = model.predict(X_test_proc)
    y_proba = model.predict_proba(X_test_proc)[:, 1]
    return accuracy_score(y_test, y_pred), roc_auc_score(y_test, y_proba)

# Collect results
rows = []
for n in n_estimators_list:
    for lr in learning_rates:
        for d in max_depths:
            acc_no, _ = eval_xgb(X_train, X_test, y_train, y_test,
                                 n, lr, d, use_pca=False)
            acc_pca, _ = eval_xgb(X_train, X_test, y_train, y_test,
                                  n, lr, d, use_pca=True)
            rows.append({
                "n_estimators": n,
                "learning_rate": lr,
                "max_depth": d,
                "Accuracy_no_PCA": acc_no,
                "Accuracy_PCA": acc_pca
            })

results_df = pd.DataFrame(rows)
print("XGBoost Performance Table")
print(results_df)

# --- Best params (No PCA) ---
best_idx = results_df['Accuracy_no_PCA'].idxmax()
best_row = results_df.iloc[best_idx]
print("\nBest Params (No PCA)")
print(best_row)

# --- ROC curve for best model ---
best_model = XGBClassifier(
    n_estimators=int(best_row["n_estimators"]),
    learning_rate=float(best_row["learning_rate"]),
    max_depth=int(best_row["max_depth"]),
    eval_metric='logloss',
    use_label_encoder=False,
    random_state=42
)
best_model.fit(X_train, y_train)
y_prob = best_model.predict_proba(X_test)[:, 1]

fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = roc_auc_score(y_test, y_prob)

plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'ROC (AUC={roc_auc:.3f})', color='crimson')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('XGBoost ROC Curve (Best Params)')
plt.legend(loc='lower right')
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [15:42:51] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [15:42:51] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [15:42:51] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [15:42:52] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:183: UserWarning: [15:42:53] WARNING: /workspace/src/learner.cc:738:
```

Stacking (base learners + meta-learner)

```
# --- Base / final models ---
svm = SVC(kernel='linear', probability=True, random_state=42)
nb  = GaussianNB()
dt  = DecisionTreeClassifier(random_state=42)
knn = KNeighborsClassifier()

log_reg = LogisticRegression(max_iter=1000, random_state=42)
rf      = RandomForestClassifier(n_estimators=100, random_state=42)

# 3 stack varieties
stacks = {
    "SVM+NB+DT → LR": StackingClassifier(
        estimators=[('svm', svm), ('nb', nb), ('dt', dt)],
        final_estimator=log_reg, passthrough=False, n_jobs=-1
    ),
    "SVM+NB+DT → RF": StackingClassifier(
        estimators=[('svm', svm), ('nb', nb), ('dt', dt)],
        final_estimator=rf, passthrough=False, n_jobs=-1
    ),
    "SVM+DT+KNN → LR": StackingClassifier(
        estimators=[('svm', svm), ('dt', dt), ('knn', knn)],
        final_estimator=log_reg, passthrough=False, n_jobs=-1
    )
```

```
}

def eval_stack(model, use_pca=False):
    if use_pca:
        pca = PCA(n_components=0.95)
        Xtr = pca.fit_transform(X_train)
        Xte = pca.transform(X_test)
    else:
        Xtr, Xte = X_train, X_test
    model.fit(Xtr, y_train)
    y_pred  = model.predict(Xte)
    y_prob  = model.predict_proba(Xte)[:, 1]
    return accuracy_score(y_test, y_pred), roc_auc_score(y_test, y_prob), y_prob

results = []
roc_curves = {}

for name, model in stacks.items():
    acc_no, auc_no, prob_no = eval_stack(model, use_pca=False)
    acc_pca, auc_pca, prob_pca = eval_stack(model, use_pca=True)
    results.append({
        "Model": name,
        "Accuracy_no_PCA": acc_no,
        "Accuracy_PCA": acc_pca,
        "AUC_no_PCA": auc_no,
        "AUC_PCA": auc_pca
```

Hyperparameter Tuning Tables

Table 1: PCA Summary

Setting	Variance Target	Explained Variance (%)	Ju
With PCA	95%	95.54	Captures 95% of total variance while reduc

Table 2: SVM - Hyperparameter Tuning Results

Kernel	C	Gamma	No-PCA	With-PCA
linear	0.1	scale	0.93135	0.924485
linear	10.0	scale	0.93135	0.924485
rbf	0.1	scale	0.93135	0.924485
rbf	10.0	scale	0.93135	0.924485

Table 3: Naive Bayes - Smoothing Choices

Smoothing	No-PCA	With-PCA
1e-09	0.720824	0.832952
1e-08	0.723112	0.832952
1e-07	0.725400	0.832952
1e-06	0.725400	0.832952

Table 4: K-Nearest Neighbors (KNN)

k	Weights	Metric	No-PCA	With-PCA
3	uniform	euclidean	0.899314	0.901602
3	distance	euclidean	0.910755	0.908467
5	distance	euclidean	0.908467	0.910755

Table 5: Logistic Regression

C	Penalty	No-PCA	With-PCA
0.01	l2	0.908467	0.908467
0.10	l1	0.913043	0.917620
1.00	l2	0.910755	0.901602

Table 6: Decision Tree

Criterion	Max Depth	No-PCA	With-PCA
gini	10	0.890160	0.874142
entropy	10	0.878719	0.878719

Table 7: Random Forest

N Estimators	Max Depth	No-PCA	With-PCA
50	10	0.924485	0.908467
100	10	0.922197	0.910755

Table 8: AdaBoost

N Estimators	Learning Rate	No-PCA	With-PCA
50	1.00	0.908467	0.892449
100	1.00	0.915332	0.901602

Table 9: Gradient Boosting

N Estimators	Learning Rate	No-PCA	With-PCA
100	0.1	0.931350	0.908467
100	0.2	0.929062	0.913043

Table 10: XGBoost

N Estimators	Learning Rate	Max Depth	No-PCA	With-PCA
100	0.1	7	0.940503	0.910755

Table 11: Stacked Models

Model	No-PCA	With-PCA
SVM+NB+DT $\rightarrow$ LR	0.917620	0.913043
SVM+DT+KNN $\rightarrow$ LR	0.919908	0.919908

Observations

- Naive Bayes and KNN benefited most from PCA. Ensemble models (RF, GB, XGB) improved moderately.
- PCA reduced variance across folds, indicating more stable results.
- PCA helped reduce overfitting in simpler models like NB and KNN.
- Linear models (SVM, Logistic Regression) showed minor performance change.
- Stacking remained robust to dimensionality reduction.

Learning Outcomes

- Learned to perform hyperparameter tuning for multiple ML classifiers.
- Applied PCA and studied its impact on model accuracy and variance.
- Identified which models benefit from PCA (e.g., NB, KNN) and which remain robust (e.g., ensembles).
- Understood that stacking ensembles maintain performance even after dimensionality reduction.

GitHub Repository: :github.com/Thamizhmathibharathi/project/tree/main/assignment6

References

1. A. Lubis, P. Sihombing, and E. Nababan, “Analysis of accuracy improvement in k-nearest neighbor using principal component analysis (PCA),” *Journal of Physics: Conference Series*, vol. 1566, p. 012062, June 2020.

2. M. Salam, A. Azar, M. Elgendy, and K. Fouad, “The effect of different dimensionality reduction techniques on machine learning overfitting problem,” *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 4, pp. 641–655, 2021.