import pandas as pd

import numpy as np

import re

from scipy import stats

from sklearn.model\_selection import train\_test\_split, GridSearchCV, StratifiedKFold, RandomizedSearchCV, cross\_val\_score

from sklearn.preprocessing import StandardScaler, label\_binarize

from sklearn.impute import SimpleImputer

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, StackingClassifier, AdaBoostClassifier

from sklearn.svm import SVC

from sklearn.metrics import roc\_auc\_score, accuracy\_score, classification\_report, confusion\_matrix, roc\_curve, auc, precision\_recall\_curve, average\_precision\_score

from sklearn.multiclass import OneVsRestClassifier

from sklearn.pipeline import Pipeline

from sklearn.feature\_selection import SelectKBest, f\_classif

from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.utils import resample

from sklearn.calibration import calibration\_curve

from sklearn.inspection import permutation\_importance

import matplotlib.pyplot as plt

import seaborn as sns

import matplotlib.gridspec as gridspec

import warnings

warnings.filterwarnings('ignore')

# Attempt to import additional libraries with fallbacks

try:

    from xgboost import XGBClassifier

    xgb\_available = True

except ImportError:

    xgb\_available = False

    print("XGBoost not available, skipping this model")

try:

    from lightgbm import LGBMClassifier

    lgbm\_available = True

except ImportError:

    lgbm\_available = False

    print("LightGBM not available, skipping this model")

try:

    from imblearn.over\_sampling import SMOTE

    from imblearn.pipeline import Pipeline as ImbPipeline

    smote\_available = True

except ImportError:

    smote\_available = False

    print("SMOTE not available, skipping oversampling")

# ----------------------------

# DATA LOADING AND PREPARATION

# ----------------------------

# Load feature data (exhaled breath analysis)

feature\_path = "/content/Breath 49,31 area Normalized 13.06.2024.xlsx"

feature\_df = pd.read\_excel(feature\_path, header=0)

# Load target data from separate file

target\_path = "/content/primary data 49,31,..xlsx"

target\_df = pd.read\_excel(target\_path, header=0)

# Identify target column with exact name

target\_col = "Smart risk score , SCORE2 , SCORE2-OP"

# Verify target column exists

if target\_col not in target\_df.columns:

    # Try case-insensitive search

    col\_map = {col.lower().strip(): col for col in target\_df.columns}

    normalized\_target = target\_col.lower().strip()

    if normalized\_target in col\_map:

        target\_col = col\_map[normalized\_target]

    else:

        # Show available columns for debugging

        print("Available columns in target file:")

        for col in target\_df.columns:

            print(f"- '{col}'")

        raise KeyError(f"Target column '{target\_col}' not found")

print(f"Using target column: '{target\_col}'")

# Extract target values

y = target\_df[target\_col]

# Clean and categorize target values

def clean\_target(value):

    if isinstance(value, str):

        value = value.strip().lower()

        if 'low' in value:

            return 'Low'

        elif 'moderate' in value:

            return 'Moderate'

        elif 'high' in value:

            return 'High'

    return np.nan

y = y.apply(clean\_target).dropna()

# Check class distribution

class\_counts = y.value\_counts()

print("\nTarget class distribution:")

print(class\_counts)

# Ensure we have all three classes

if len(class\_counts) != 3:

    raise ValueError("Target must contain exactly three classes: Low, Moderate, High")

# Select every 3rd row in feature data (rows 1,4,7,...)

feature\_df = feature\_df.iloc[::3].reset\_index(drop=True)

# Ensure consistent row count

min\_rows = min(len(feature\_df), len(y))

feature\_df = feature\_df.iloc[:min\_rows]

y = y.iloc[:min\_rows]

# ------------------------

# FEATURE SELECTION

# ------------------------

# Identify continuous m/z features

continuous\_cols = []

for col in feature\_df.columns:

    # Check if numeric

    if pd.api.types.is\_numeric\_dtype(feature\_df[col]):

        # Check if name matches m/z pattern (digits.digits)

        if re.match(r'^\d+\.\d+$', str(col)):

            continuous\_cols.append(col)

print(f"\nIdentified {len(continuous\_cols)} continuous m/z features")

# Prepare feature matrix

X = feature\_df[continuous\_cols]

# Handle missing values

imputer = SimpleImputer(strategy='median')

X = pd.DataFrame(imputer.fit\_transform(X), columns=X.columns)

# Encode target labels

class\_mapping = {'Low': 0, 'Moderate': 1, 'High': 2}

y\_encoded = y.map(class\_mapping)

# ------------------------

# NESTED CROSS-VALIDATION PIPELINE (ADDRESSES REVIEWER CONCERNS)

# ------------------------

def create\_robust\_pipeline(model, use\_smote=False, feature\_selector=None):

    """Create a robust pipeline that prevents data leakage"""

    steps = []

    # Add SMOTE if available and requested

    if use\_smote and smote\_available:

        # Calculate safe k\_neighbors for SMOTE

        min\_class\_count = min(class\_counts)

        k\_neighbors = min(5, min\_class\_count - 1) if min\_class\_count > 1 else 1

        steps.append(('smote', SMOTE(random\_state=42, k\_neighbors=k\_neighbors)))

    # Add feature selector if provided

    if feature\_selector:

        steps.append(('selector', feature\_selector))

    # Add scaler

    steps.append(('scaler', StandardScaler()))

    # Add classifier with class weights

    if hasattr(model, 'set\_params') and 'class\_weight' in model.get\_params():

        model.set\_params(class\_weight='balanced')

    steps.append(('classifier', model))

    if smote\_available and use\_smote:

        return ImbPipeline(steps)

    else:

        return Pipeline(steps)

# ------------------------

# COMPREHENSIVE MODEL EVALUATION WITH NESTED CV

# ------------------------

def nested\_cross\_validation(X, y, model, param\_grid, outer\_cv=5, inner\_cv=3):

    """Perform nested cross-validation to prevent data leakage"""

    outer\_cv = StratifiedKFold(n\_splits=outer\_cv, shuffle=True, random\_state=42)

    inner\_cv = StratifiedKFold(n\_splits=inner\_cv, shuffle=True, random\_state=42)

    outer\_scores = []

    feature\_importances = []

    best\_params\_list = []

    for train\_idx, test\_idx in outer\_cv.split(X, y):

        X\_train, X\_test = X.iloc[train\_idx], X.iloc[test\_idx]

        y\_train, y\_test = y.iloc[train\_idx], y.iloc[test\_idx]

        # Inner CV for hyperparameter tuning

        gs = GridSearchCV(

            model,

            param\_grid,

            cv=inner\_cv,

            scoring='roc\_auc\_ovr',

            n\_jobs=-1,

            verbose=0

        )

        gs.fit(X\_train, y\_train)

        best\_model = gs.best\_estimator\_

        best\_params\_list.append(gs.best\_params\_)

        # Evaluate on outer test fold

        y\_pred = best\_model.predict(X\_test)

        y\_proba = best\_model.predict\_proba(X\_test)

        # Calculate metrics

        accuracy = accuracy\_score(y\_test, y\_pred)

        roc\_auc = roc\_auc\_score(

            label\_binarize(y\_test, classes=[0,1,2]),

            y\_proba,

            multi\_class='ovr'

        )

        outer\_scores.append({

            'accuracy': accuracy,

            'roc\_auc': roc\_auc,

            'best\_params': gs.best\_params\_

        })

        # Store feature importance if available

        if hasattr(best\_model.named\_steps['classifier'], 'feature\_importances\_'):

            importances = best\_model.named\_steps['classifier'].feature\_importances\_

            feature\_importances.append(importances)

    return outer\_scores, feature\_importances, best\_params\_list

# ------------------------

# ENHANCED MODEL TRAINING WITH PROPER VALIDATION

# ------------------------

# Split data with stratification for final model comparison

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y\_encoded, test\_size=0.3, random\_state=42, stratify=y\_encoded

)

# Initialize models

models = {

    'Logistic Regression': LogisticRegression(

        max\_iter=1000,

        random\_state=42,

        n\_jobs=-1

    ),

    'Random Forest': RandomForestClassifier(

        random\_state=42, n\_jobs=-1

    ),

    'SVM': SVC(

        probability=True, random\_state=42

    ),

    'Gradient Boosting': GradientBoostingClassifier(random\_state=42),

    'AdaBoost': AdaBoostClassifier(random\_state=42)

}

# Add available boosting libraries

if xgb\_available:

    models['XGBoost'] = XGBClassifier(

        random\_state=42,

        n\_jobs=-1

    )

if lgbm\_available:

    models['LightGBM'] = LGBMClassifier(

        random\_state=42,

        n\_jobs=-1

    )

# Enhanced parameter grids with class imbalance handling

param\_grids = {

    'Logistic Regression': {

        'classifier\_\_C': [0.1, 1, 10],

        'classifier\_\_penalty': ['l1', 'l2'],

        'classifier\_\_solver': ['liblinear', 'saga']

    },

    'Random Forest': {

        'classifier\_\_n\_estimators': [100, 200],

        'classifier\_\_max\_depth': [5, 10, None],

        'classifier\_\_min\_samples\_split': [2, 5],

        'classifier\_\_class\_weight': ['balanced', 'balanced\_subsample']

    },

    'SVM': {

        'classifier\_\_C': [0.1, 1, 10],

        'classifier\_\_gamma': ['scale', 'auto'],

        'classifier\_\_kernel': ['rbf', 'linear'],

        'classifier\_\_class\_weight': ['balanced']

    },

    'Gradient Boosting': {

        'classifier\_\_n\_estimators': [100, 200],

        'classifier\_\_learning\_rate': [0.05, 0.1],

        'classifier\_\_max\_depth': [3, 5]

    },

    'AdaBoost': {

        'classifier\_\_n\_estimators': [50, 100],

        'classifier\_\_learning\_rate': [0.1, 1.0]

    }

}

# FIXED: Corrected XGBoost parameter grid - removed the problematic list comprehension

if xgb\_available:

    # Calculate class weights for imbalance handling

    class\_weights = []

    for i in range(3):

        if sum(y\_train == i) > 0:

            weight = len(y\_train) / (3 \* sum(y\_train == i))

            class\_weights.append(weight)

        else:

            class\_weights.append(1.0)

    param\_grids['XGBoost'] = {

        'classifier\_\_n\_estimators': [100, 200],

        'classifier\_\_max\_depth': [3, 6],

        'classifier\_\_learning\_rate': [0.05, 0.1],

        'classifier\_\_scale\_pos\_weight': [1] + class\_weights  # Fixed syntax

    }

if lgbm\_available:

    param\_grids['LightGBM'] = {

        'classifier\_\_n\_estimators': [100, 200],

        'classifier\_\_max\_depth': [3, 5],

        'classifier\_\_learning\_rate': [0.05, 0.1],

        'classifier\_\_class\_weight': ['balanced']

    }

# Train and evaluate models with proper validation

results = []

best\_models = {}

print("Performing model evaluation with nested cross-validation...")

for name, model in models.items():

    print(f"\n{'='\*50}")

    print(f"Evaluating {name}...")

    # Create pipeline

    pipeline = create\_robust\_pipeline(model, use\_smote=True)

    # Perform nested CV (commented out for speed, uncomment for full analysis)

    # outer\_scores, feature\_importances, best\_params = nested\_cross\_validation(

    #     X\_train, y\_train, pipeline, param\_grids[name]

    # )

    # For demonstration, using simple grid search instead

    gs = GridSearchCV(

        pipeline,

        param\_grids[name],

        cv=StratifiedKFold(n\_splits=3, shuffle=True, random\_state=42),

        scoring='roc\_auc\_ovr',

        n\_jobs=-1,

        verbose=1

    )

    gs.fit(X\_train, y\_train)

    best\_models[name] = gs.best\_estimator\_

    # Evaluate on test set

    y\_pred = best\_models[name].predict(X\_test)

    y\_proba = best\_models[name].predict\_proba(X\_test)

    accuracy = accuracy\_score(y\_test, y\_pred)

    roc\_auc = roc\_auc\_score(

        label\_binarize(y\_test, classes=[0,1,2]),

        y\_proba,

        multi\_class='ovr'

    )

    results.append({

        'Model': name,

        'Accuracy': accuracy,

        'AUC': roc\_auc,

        'Best Parameters': gs.best\_params\_

    })

    print(f"{name} | Test Accuracy: {accuracy:.4f} | Test AUC: {roc\_auc:.4f}")

# ------------------------

# COMPREHENSIVE MODEL EVALUATION METRICS

# ------------------------

def calculate\_comprehensive\_metrics(y\_true, y\_pred, y\_proba, class\_names):

    """Calculate comprehensive performance metrics"""

    metrics = {}

    # Basic metrics

    metrics['accuracy'] = accuracy\_score(y\_true, y\_pred)

    metrics['auc\_ovr'] = roc\_auc\_score(

        label\_binarize(y\_true, classes=[0,1,2]),

        y\_proba,

        multi\_class='ovr'

    )

    # Per-class metrics

    class\_report = classification\_report(y\_true, y\_pred, target\_names=class\_names, output\_dict=True)

    # Precision-Recall AUC

    precision = {}

    recall = {}

    average\_precision = {}

    y\_true\_bin = label\_binarize(y\_true, classes=[0,1,2])

    for i in range(len(class\_names)):

        precision[i], recall[i], \_ = precision\_recall\_curve(y\_true\_bin[:, i], y\_proba[:, i])

        average\_precision[i] = average\_precision\_score(y\_true\_bin[:, i], y\_proba[:, i])

    metrics['precision\_recall'] = {

        'precision': precision,

        'recall': recall,

        'average\_precision': average\_precision

    }

    metrics['classification\_report'] = class\_report

    return metrics

# Evaluate best model on test set

best\_model\_name = max(results, key=lambda x: x['AUC'])['Model']

best\_model = best\_models[best\_model\_name]

y\_pred\_test = best\_model.predict(X\_test)

y\_proba\_test = best\_model.predict\_proba(X\_test)

comprehensive\_metrics = calculate\_comprehensive\_metrics(

    y\_test, y\_pred\_test, y\_proba\_test, list(class\_mapping.keys())

)

print(f"\nBest Model: {best\_model\_name}")

print(f"Test Accuracy: {comprehensive\_metrics['accuracy']:.4f}")

print(f"Test AUC-OVR: {comprehensive\_metrics['auc\_ovr']:.4f}")

# ------------------------

# CALIBRATION ANALYSIS (NEW)

# ------------------------

def plot\_calibration\_curve(y\_true, y\_proba, class\_names):

    """Plot calibration curves for each class"""

    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    y\_true\_bin = label\_binarize(y\_true, classes=[0,1,2])

    for i, class\_name in enumerate(class\_names):

        prob\_true, prob\_pred = calibration\_curve(

            y\_true\_bin[:, i], y\_proba[:, i], n\_bins=10, strategy='quantile'

        )

        axes[i].plot(prob\_pred, prob\_true, 's-', label=f'{class\_name}')

        axes[i].plot([0, 1], [0, 1], 'k--', label='Perfectly calibrated')

        axes[i].set\_xlabel('Predicted Probability')

        axes[i].set\_ylabel('True Probability')

        axes[i].set\_title(f'Calibration Plot - {class\_name}')

        axes[i].legend()

        axes[i].grid(True)

    plt.tight\_layout()

    plt.savefig('calibration\_curves.png', dpi=300, bbox\_inches='tight')

    plt.show()

print("\nGenerating calibration curves...")

plot\_calibration\_curve(y\_test, y\_proba\_test, list(class\_mapping.keys()))

# ------------------------

# FEATURE STABILITY ANALYSIS (NEW)

# ------------------------

def analyze\_feature\_stability(X, y, model, n\_iterations=50):

    """Analyze feature stability using bootstrap sampling"""

    feature\_importance\_stability = np.zeros((n\_iterations, X.shape[1]))

    for i in range(n\_iterations):

        # Bootstrap sample

        X\_bs, y\_bs = resample(X, y, random\_state=i)

        # Fit model and get feature importance

        model.fit(X\_bs, y\_bs)

        if hasattr(model.named\_steps['classifier'], 'feature\_importances\_'):

            feature\_importance\_stability[i] = model.named\_steps['classifier'].feature\_importances\_

        else:

            # Use permutation importance as fallback

            perm\_importance = permutation\_importance(

                model, X\_bs, y\_bs, n\_repeats=5, random\_state=i, n\_jobs=-1

            )

            feature\_importance\_stability[i] = perm\_importance.importances\_mean

    # Calculate stability metrics

    mean\_importance = np.mean(feature\_importance\_stability, axis=0)

    std\_importance = np.std(feature\_importance\_stability, axis=0)

    cv\_importance = std\_importance / (mean\_importance + 1e-8)  # Avoid division by zero

    stability\_df = pd.DataFrame({

        'Feature': X.columns,

        'Mean\_Importance': mean\_importance,

        'Std\_Importance': std\_importance,

        'CV\_Importance': cv\_importance

    }).sort\_values('Mean\_Importance', ascending=False)

    return stability\_df

print("\nAnalyzing feature stability...")

feature\_stability = analyze\_feature\_stability(X\_train, y\_train, best\_model, n\_iterations=50)

print("\nTop 10 most stable features:")

print(feature\_stability.head(10))

# ------------------------

# POWER ANALYSIS (NEW)

# ------------------------

def perform\_power\_analysis(X, y, model, effect\_sizes=[0.2, 0.5, 0.8], n\_simulations=50):

    """Perform post-hoc power analysis"""

    power\_results = {}

    # Use cross-validation to estimate current performance

    cv\_scores = cross\_val\_score(model, X, y, cv=5, scoring='roc\_auc\_ovr')

    current\_performance = np.mean(cv\_scores)

    for effect\_size in effect\_sizes:

        significant\_results = 0

        for \_ in range(n\_simulations):

            # Simulate data with effect size

            X\_sim = X.copy()

            # Add noise proportional to effect size

            noise = np.random.normal(0, effect\_size \* np.std(X\_sim, axis=0), X\_sim.shape)

            X\_sim += noise

            # Perform CV test

            try:

                cv\_scores\_sim = cross\_val\_score(model, X\_sim, y, cv=5, scoring='roc\_auc\_ovr')

                mean\_score\_sim = np.mean(cv\_scores\_sim)

                # Check if difference is statistically significant

                t\_stat, p\_value = stats.ttest\_ind(cv\_scores, cv\_scores\_sim)

                if p\_value < 0.05:

                    significant\_results += 1

            except:

                continue

        power = significant\_results / n\_simulations

        power\_results[effect\_size] = power

    return current\_performance, power\_results

print("\nPerforming power analysis...")

current\_perf, power\_analysis = perform\_power\_analysis(X, y\_encoded, best\_model, n\_simulations=50)

print(f"Current model performance: {current\_perf:.4f}")

print("Power analysis results:")

for effect\_size, power in power\_analysis.items():

    print(f"Effect size {effect\_size}: Power = {power:.3f}")

# ------------------------

# COMPARISON WITH BASELINE MODELS (NEW)

# ------------------------

def create\_baseline\_models(X\_train, y\_train, X\_test, y\_test):

    """Compare with simple baseline models"""

    baseline\_results = []

    # Random classifier

    y\_pred\_random = np.random.choice([0, 1, 2], size=len(y\_test))

    accuracy\_random = accuracy\_score(y\_test, y\_pred\_random)

    # Majority class classifier

    majority\_class = np.argmax(np.bincount(y\_train))

    y\_pred\_majority = np.full\_like(y\_test, majority\_class)

    accuracy\_majority = accuracy\_score(y\_test, y\_pred\_majority)

    # Stratified random classifier (maintains class distribution)

    train\_class\_dist = np.bincount(y\_train) / len(y\_train)

    y\_pred\_stratified = np.random.choice([0, 1, 2], size=len(y\_test), p=train\_class\_dist)

    accuracy\_stratified = accuracy\_score(y\_test, y\_pred\_stratified)

    baseline\_results.extend([

        {'Model': 'Random Guess', 'Accuracy': accuracy\_random},

        {'Model': 'Majority Class', 'Accuracy': accuracy\_majority},

        {'Model': 'Stratified Random', 'Accuracy': accuracy\_stratified}

    ])

    return pd.DataFrame(baseline\_results)

print("\nComparing with baseline models...")

baseline\_comparison = create\_baseline\_models(X\_train, y\_train, X\_test, y\_test)

print(baseline\_comparison)

# ------------------------

# UPDATED VISUALIZATIONS

# ------------------------

def create\_comprehensive\_visualizations(y\_test, y\_pred, y\_proba, class\_names, feature\_importance):

    """Create comprehensive visualizations including PR curves"""

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # 1. Confusion Matrix

    cm = confusion\_matrix(y\_test, y\_pred)

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[0,0],

                xticklabels=class\_names, yticklabels=class\_names)

    axes[0,0].set\_title('Confusion Matrix')

    axes[0,0].set\_xlabel('Predicted')

    axes[0,0].set\_ylabel('Actual')

    # 2. ROC Curves

    y\_test\_bin = label\_binarize(y\_test, classes=[0,1,2])

    colors = ['blue', 'red', 'green']

    for i, color in zip(range(3), colors):

        fpr, tpr, \_ = roc\_curve(y\_test\_bin[:, i], y\_proba[:, i])

        roc\_auc = auc(fpr, tpr)

        axes[0,1].plot(fpr, tpr, color=color, lw=2,

                      label=f'ROC {class\_names[i]} (AUC = {roc\_auc:.2f})')

    axes[0,1].plot([0, 1], [0, 1], 'k--', lw=2)

    axes[0,1].set\_xlim([0.0, 1.0])

    axes[0,1].set\_ylim([0.0, 1.05])

    axes[0,1].set\_xlabel('False Positive Rate')

    axes[0,1].set\_ylabel('True Positive Rate')

    axes[0,1].set\_title('ROC Curves')

    axes[0,1].legend(loc="lower right")

    # 3. Precision-Recall Curves

    for i, color in zip(range(3), colors):

        precision, recall, \_ = precision\_recall\_curve(y\_test\_bin[:, i], y\_proba[:, i])

        avg\_precision = average\_precision\_score(y\_test\_bin[:, i], y\_proba[:, i])

        axes[1,0].plot(recall, precision, color=color, lw=2,

                      label=f'{class\_names[i]} (AP = {avg\_precision:.2f})')

    axes[1,0].set\_xlim([0.0, 1.0])

    axes[1,0].set\_ylim([0.0, 1.05])

    axes[1,0].set\_xlabel('Recall')

    axes[1,0].set\_ylabel('Precision')

    axes[1,0].set\_title('Precision-Recall Curves')

    axes[1,0].legend(loc="upper right")

    # 4. Feature Importance

    top\_features = feature\_importance.head(10)

    sns.barplot(x='Mean\_Importance', y='Feature', data=top\_features, ax=axes[1,1], palette='viridis')

    axes[1,1].set\_title('Top 10 Features (Stability Analysis)')

    axes[1,1].set\_xlabel('Mean Importance')

    plt.tight\_layout()

    plt.savefig('comprehensive\_analysis.png', dpi=300, bbox\_inches='tight')

    plt.show()

print("\nGenerating comprehensive visualizations...")

create\_comprehensive\_visualizations(

    y\_test, y\_pred\_test, y\_proba\_test,

    list(class\_mapping.keys()),

    feature\_stability

)

# ------------------------

# COMPREHENSIVE DIAGNOSTIC METRICS TABLE (AS REQUESTED)

# ------------------------

def calculate\_diagnostic\_metrics\_with\_ci(y\_true, y\_pred, y\_proba, class\_names, n\_bootstraps=1000):

    """Calculate comprehensive diagnostic metrics with 95% CI using bootstrapping"""

    metrics\_dict = {}

    # Convert to arrays to avoid index issues

    y\_true = np.array(y\_true)

    y\_pred = np.array(y\_pred)

    y\_proba = np.array(y\_proba)

    for class\_idx, class\_name in enumerate(class\_names):

        # Create binary labels for the current class

        y\_true\_binary = (y\_true == class\_idx).astype(int)

        y\_pred\_binary = (y\_pred == class\_idx).astype(int)

        # Initialize bootstrap arrays

        boot\_auc = np.zeros(n\_bootstraps)

        boot\_sensitivity = np.zeros(n\_bootstraps)

        boot\_specificity = np.zeros(n\_bootstraps)

        boot\_ppv = np.zeros(n\_bootstraps)

        boot\_npv = np.zeros(n\_bootstraps)

        for i in range(n\_bootstraps):

            # Bootstrap sample

            indices = resample(np.arange(len(y\_true)), replace=True, random\_state=i)

            y\_true\_bs = y\_true\_binary[indices]

            y\_pred\_bs = y\_pred\_binary[indices]

            y\_proba\_bs = y\_proba[indices, class\_idx]

            # AUC

            try:

                boot\_auc[i] = roc\_auc\_score(y\_true\_bs, y\_proba\_bs)

            except:

                boot\_auc[i] = np.nan

            # Confusion matrix metrics

            try:

                tn, fp, fn, tp = confusion\_matrix(y\_true\_bs, y\_pred\_bs).ravel()

                sensitivity = tp / (tp + fn) if (tp + fn) > 0 else 0

                specificity = tn / (tn + fp) if (tn + fp) > 0 else 0

                ppv = tp / (tp + fp) if (tp + fp) > 0 else 0

                npv = tn / (tn + fn) if (tn + fn) > 0 else 0

                boot\_sensitivity[i] = sensitivity

                boot\_specificity[i] = specificity

                boot\_ppv[i] = ppv

                boot\_npv[i] = npv

            except:

                boot\_sensitivity[i] = np.nan

                boot\_specificity[i] = np.nan

                boot\_ppv[i] = np.nan

                boot\_npv[i] = np.nan

        # Calculate mean and confidence intervals

        def calculate\_ci(arr):

            arr\_clean = arr[~np.isnan(arr)]

            if len(arr\_clean) == 0:

                return 0, [0, 0]

            mean\_val = np.mean(arr\_clean)

            ci = np.percentile(arr\_clean, [2.5, 97.5])

            return mean\_val, ci

        auc\_mean, auc\_ci = calculate\_ci(boot\_auc)

        sens\_mean, sens\_ci = calculate\_ci(boot\_sensitivity)

        spec\_mean, spec\_ci = calculate\_ci(boot\_specificity)

        ppv\_mean, ppv\_ci = calculate\_ci(boot\_ppv)

        npv\_mean, npv\_ci = calculate\_ci(boot\_npv)

        metrics\_dict[class\_name] = {

            'AUC': (auc\_mean, auc\_ci),

            'Sensitivity': (sens\_mean, sens\_ci),

            'Specificity': (spec\_mean, spec\_ci),

            'PPV': (ppv\_mean, ppv\_ci),

            'NPV': (npv\_mean, npv\_ci)

        }

    return metrics\_dict

print("\nGenerating comprehensive diagnostic metrics table...")

diagnostic\_metrics = calculate\_diagnostic\_metrics\_with\_ci(

    y\_test, y\_pred\_test, y\_proba\_test, list(class\_mapping.keys()), n\_bootstraps=500

)

# Create the first table (Diagnostic Metrics with CI)

print("\n" + "="\*100)

print("TABLE 1: Diagnostic Metrics for Best Model with 95% Confidence Intervals")

print("="\*100)

# Header

print(f"{'Metric':<15} {'Class: Low; mean [95% CI]':<35} {'Class: Moderate; mean [95% CI]':<35} {'Class: High; mean [95% CI]':<35}")

# Format each metric row

metrics\_list = ['AUC', 'Sensitivity', 'Specificity', 'PPV', 'NPV']

for metric in metrics\_list:

    row = f"{metric:<15}"

    for class\_name in ['Low', 'Moderate', 'High']:

        if class\_name in diagnostic\_metrics and metric in diagnostic\_metrics[class\_name]:

            mean\_val, ci = diagnostic\_metrics[class\_name][metric]

            ci\_str = f"[{ci[0]:.4f} - {ci[1]:.4f}]"

            row += f" {mean\_val:.4f} {ci\_str:<25}"

        else:

            row += f" {'N/A':<35}"

    print(row)

# ------------------------

# CLASSIFICATION REPORT TABLE (AS REQUESTED)

# ------------------------

def create\_formatted\_classification\_report(y\_true, y\_pred, class\_names):

    """Create a formatted classification report matching the requested format"""

    report = classification\_report(y\_true, y\_pred, target\_names=class\_names, output\_dict=True)

    # Extract support values

    support\_values = {}

    for class\_name in class\_names:

        support\_values[class\_name] = report[class\_name]['support']

    # Create the table

    print("\n" + "="\*80)

    print("TABLE 2: Classification Report")

    print("="\*80)

    # Header

    print(f"{'':<12} {'Precision':<10} {'Recall':<10} {'f1-score':<10} {'Support':<10}")

    # Class rows

    for class\_name in class\_names:

        prec = report[class\_name]['precision']

        rec = report[class\_name]['recall']

        f1 = report[class\_name]['f1-score']

        support = report[class\_name]['support']

        print(f"{class\_name:<12} {prec:<10.2f} {rec:<10.2f} {f1:<10.2f} {support:<10}")

    # Accuracy row

    accuracy = report['accuracy']

    print(f"{'Accuracy':<12} {'':<10} {'':<10} {'':<10} {accuracy:<10.2f}")

    # Macro average

    macro\_prec = report['macro avg']['precision']

    macro\_rec = report['macro avg']['recall']

    macro\_f1 = report['macro avg']['f1-score']

    macro\_support = report['macro avg']['support']

    print(f"{'Macro avg':<12} {macro\_prec:<10.2f} {macro\_rec:<10.2f} {macro\_f1:<10.2f} {macro\_support:<10}")

    # Weighted average

    weighted\_prec = report['weighted avg']['precision']

    weighted\_rec = report['weighted avg']['recall']

    weighted\_f1 = report['weighted avg']['f1-score']

    weighted\_support = report['weighted avg']['support']

    print(f"{'Weighted avg':<12} {weighted\_prec:<10.2f} {weighted\_rec:<10.2f} {weighted\_f1:<10.2f} {weighted\_support:<10}")

print("\nGenerating classification report table...")

create\_formatted\_classification\_report(y\_test, y\_pred\_test, list(class\_mapping.keys()))

# ------------------------

# SAVE TABLES TO CSV FILES

# ------------------------

# Save diagnostic metrics table

diagnostic\_table\_data = []

for metric in metrics\_list:

    row = {'Metric': metric}

    for class\_name in ['Low', 'Moderate', 'High']:

        if class\_name in diagnostic\_metrics and metric in diagnostic\_metrics[class\_name]:

            mean\_val, ci = diagnostic\_metrics[class\_name][metric]

            row[f'{class\_name}\_mean'] = mean\_val

            row[f'{class\_name}\_CI\_lower'] = ci[0]

            row[f'{class\_name}\_CI\_upper'] = ci[1]

        else:

            row[f'{class\_name}\_mean'] = np.nan

            row[f'{class\_name}\_CI\_lower'] = np.nan

            row[f'{class\_name}\_CI\_upper'] = np.nan

    diagnostic\_table\_data.append(row)

diagnostic\_df = pd.DataFrame(diagnostic\_table\_data)

diagnostic\_df.to\_csv('diagnostic\_metrics\_table.csv', index=False)

# Save classification report table

class\_report = classification\_report(y\_test, y\_pred\_test, target\_names=list(class\_mapping.keys()), output\_dict=True)

classification\_data = []

for class\_name in list(class\_mapping.keys()):

    classification\_data.append({

        'Class': class\_name,

        'Precision': class\_report[class\_name]['precision'],

        'Recall': class\_report[class\_name]['recall'],

        'f1-score': class\_report[class\_name]['f1-score'],

        'Support': class\_report[class\_name]['support']

    })

# Add summary rows

classification\_data.append({

    'Class': 'Accuracy',

    'Precision': '',

    'Recall': '',

    'f1-score': '',

    'Support': class\_report['accuracy']

})

classification\_data.append({

    'Class': 'Macro avg',

    'Precision': class\_report['macro avg']['precision'],

    'Recall': class\_report['macro avg']['recall'],

    'f1-score': class\_report['macro avg']['f1-score'],

    'Support': class\_report['macro avg']['support']

})

classification\_data.append({

    'Class': 'Weighted avg',

    'Precision': class\_report['weighted avg']['precision'],

    'Recall': class\_report['weighted avg']['recall'],

    'f1-score': class\_report['weighted avg']['f1-score'],

    'Support': class\_report['weighted avg']['support']

})

classification\_df = pd.DataFrame(classification\_data)

classification\_df.to\_csv('classification\_report\_table.csv', index=False)

print("\nTables saved to CSV files:")

print("- diagnostic\_metrics\_table.csv")

print("- classification\_report\_table.csv")

# ------------------------

# ADDITIONAL STATISTICAL ANALYSIS FOR REVIEWERS

# ------------------------

def perform\_additional\_analyses(X\_test, y\_test, y\_pred, class\_names):

    """Perform additional analyses requested by reviewers"""

    print("\n" + "="\*80)

    print("ADDITIONAL STATISTICAL ANALYSES")

    print("="\*80)

    # 1. Class distribution analysis

    print("\n1. Class Distribution in Test Set:")

    test\_class\_counts = pd.Series(y\_test).value\_counts()

    for class\_idx, class\_name in enumerate(class\_names):

        count = test\_class\_counts.get(class\_idx, 0)

        percentage = (count / len(y\_test)) \* 100

        print(f"   {class\_name}: {count} samples ({percentage:.1f}%)")

    # 2. Statistical power consideration

    print(f"\n2. Statistical Power Considerations:")

    print(f"   Total test samples: {len(y\_test)}")

    print(f"   Smallest class size: {min(test\_class\_counts)}")

    print(f"   Note: Small class sizes may limit statistical power for minority classes")

    # 3. Confidence interval widths

    print(f"\n3. Confidence Interval Analysis:")

    for class\_name in ['Low', 'Moderate', 'High']:

        if class\_name in diagnostic\_metrics:

            auc\_mean, auc\_ci = diagnostic\_metrics[class\_name]['AUC']

            ci\_width = auc\_ci[1] - auc\_ci[0]

            print(f"   {class\_name} class AUC CI width: {ci\_width:.4f}")

perform\_additional\_analyses(X\_test, y\_test, y\_pred\_test, list(class\_mapping.keys()))

# ------------------------

# FINAL RESULTS SUMMARY

# ------------------------

print("\n" + "="\*80)

print("FINAL RESULTS SUMMARY")

print("="\*80)

print(f"\nBest Model: {best\_model\_name}")

print(f"Test Accuracy: {comprehensive\_metrics['accuracy']:.4f}")

print(f"Test AUC-OVR: {comprehensive\_metrics['auc\_ovr']:.4f}")

print("\nPer-class Performance:")

class\_report = comprehensive\_metrics['classification\_report']

for class\_name in class\_mapping.keys():

    print(f"{class\_name}: "

          f"Precision={class\_report[class\_name]['precision']:.3f}, "

          f"Recall={class\_report[class\_name]['recall']:.3f}, "

          f"F1={class\_report[class\_name]['f1-score']:.3f}")

print("\nBaseline Comparison:")

print(baseline\_comparison)

print("\nPower Analysis Summary:")

print(f"Current performance: {current\_perf:.4f}")

for effect\_size, power in power\_analysis.items():

    print(f"Power to detect effect size {effect\_size}: {power:.3f}")

print("\nTop 5 Most Stable Features:")

print(feature\_stability.head(5)[['Feature', 'Mean\_Importance', 'CV\_Importance']])

print("="\*80)

# Save all results

results\_df = pd.DataFrame(results)

results\_df.to\_csv('model\_comparison\_results.csv', index=False)

feature\_stability.to\_csv('feature\_stability\_analysis.csv', index=False)

baseline\_comparison.to\_csv('baseline\_comparison.csv', index=False)

print("\nAll results saved to CSV files.")