Hybrid Water Quality Assessment Using K-Means, Random Forest, SVM, and k-NN: A Case Study of the Mahakali River, Nepal

Tek Raj Bhatt
Student ID: 250069, *CUID: 16544288*
MSc. Data Science and Computational Intelligence
Softwarica College of IT and E-commerce (Coventry University)
250069@softwarica.edu.np

***Abstract*—Water quality assessment is critical for sustainable water resource management in developing regions. This study analyses the water quality in the Mahakali River basin of Nepal using a hybrid machine learning approach that combines supervised and unsupervised learning. Over the course of three years (2023–2025), bi-monthly measurements from fifteen monitoring stations produced 720 samples that were characterised by ten physicochemical parameters. Principal Component Analysis decreased the dimensionality from 10 to 7 components while keeping 92.75% of the variance after K-Nearest Neighbours imputation for missing values (12.5%). Two different regimes were found using K-means clustering: Cluster 0 (38%, "Poor" quality, WQI=62.90) and Cluster 1 (62%, "Good" quality, WQI=36.54). Three supervised classifiers performed exceptionally well: Random Forest (97.92%), SVM (97.22%), and k-NN (99.31% accuracy). Ammonia (32.18%), iron (20.23%), and chloride (13.22%) were identified as the top predictors by feature importance analysis. This study shows that hybrid machine learning techniques can provide actionable insights for focused interventions and characterise water quality with 99% accuracy.**

***Index Terms*—water quality assessment, machine learning, K-means clustering, Random Forest, Support Vector Machine, k-Nearest Neighbors, Water Quality Index, Mahakali River, Nepal**

I. Introduction

Water quality is a fundamental determinant of public health and sustainable development. Freshwater resources in developing countries are increasingly contaminated by domestic sewage, industrial effluents, and agricultural runoff [2]. Nepal has a lot of water from rivers that are fed by snow, but the water quality is bad because the treatment infrastructure is not good enough [3].

The Mahakali River is an important source of fresh water for Nepal's Far-Western Province. It provides drinking water, irrigation, and hydropower to more than 500,000 people living in the Baitadi, Dadeldhura, and Kanchanpur districts. However, increasing anthropogenic activities have raised concerns about deteriorating water quality. Traditional monitoring approaches rely on periodic sampling, generating discrete snapshots that may not capture temporal dynamics or identify patterns in complex multivariate datasets.

Machine learning techniques possess the capacity to revolutionise environmental monitoring through the extraction of patterns from high-dimensional data and the facilitation of predictive modelling [1], [4]. Unsupervised learning algorithms effectively identify natural groupings in the absence of predefined labels, whereas supervised methods attain high predictive accuracy in classification tasks [5].

This study uses a hybrid methodology that integrates Principal Component Analysis (PCA), K-means clustering, Water Quality Index (WQI) computation, and three supervised classifiers—Random Forest (RF), Support Vector Machine (SVM), and k-Nearest Neighbours (k-NN)—for the evaluation of water quality in the Mahakali River basin.

The main goal is to create and test a hybrid machine learning method that can classify data with more than 80% accuracy. Secondary objectives encompass dataset preprocessing, the application of PCA for dimensionality reduction, the implementation of K-means clustering, the calculation of WQI for objective labelling, the training of supervised classifiers, and the analysis of spatial-temporal patterns.

II. The Data Set

Oxfam's TROSA programmes, which took place every two months from January 2023 to December 2025 at 15 monitoring stations in three districts, provided data on water quality. The basin has elevations ranging from 150 m to more than 7,000 m above sea level. It has subtropical to alpine climates, with an average of 1,500 to 2,500 mm of rain falling each year, mostly during the monsoon season.

Samples were taken according to WHO guidelines [3], and they were tested at certified labs within 24 hours. We measured ten physicochemical parameters: temperature, pH, total hardness, chloride, ammonia, phosphate, nitrate, iron, free residual chlorine, and coliform bacteria. Table I gives a short summary of the dataset's features.

III. Data Preparation

A. Data Preprocessing Pipeline

Raw monitoring data required extensive preprocessing addressing quality issues inherent in field-collected environ-

TABLE I
DATASET CHARACTERISTICS AND STRUCTURE

Characteristic	Description
Temporal coverage	January 2023 - December 2025 (3 years)
Sampling frequency	Bi-monthly (6 samples/station/year)
Spatial coverage	15 stations across 3 districts
Total samples	720
Parameters	10 physicochemical indicators

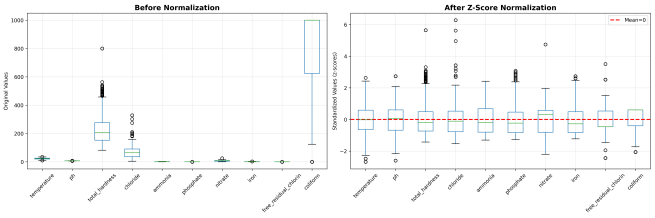

Fig. 1. Comparison of feature distribution before and after Z-score normalisation, showing all parameters transformed to zero mean and unit variance.

mental datasets. The six-stage pipeline included: (1) data loading and structural validation, (2) metadata integration, (3) missing value analysis showing 898 missing values (12.47%), (4) text normalisation turning qualitative coliform observations into numbers, (5) K-Nearest Neighbours imputation with $\mathrm{n}_{neighbors} = 5, and (6) feature scaling using Z-score normalisation : z_i = \frac{x_i - \mu}{\sigma}$(1) where x_i is the original value, μ is the mean, and σ is the standard deviation.

B. Class Imbalance Handling

The dataset had a moderate class imbalance, with 61.67% of the data being good and 38.33% being bad (a ratio of 1.61:1). To avoid bias towards the majority class, the Synthetic Minority Over-sampling Technique (SMOTE) was used to make the class distribution in the training set 50:50 [6].

C. Dimensionality Reduction

Principal Component Analysis was applied for dimensionality reduction. The mathematical formulation involves eigendecomposition of the covariance matrix:

$$\mathbf{C} = \frac{1}{n-1}\mathbf{X}^T\mathbf{X} \tag{2}$$

where $\mathbf{C}$ is the covariance matrix and $\mathbf{X}$ is the standardized data matrix.

PCA analysis identified that the first seven components cumulatively explained 92.75% of total variance. PC1 dominated variance structure (49.34%) with high positive loadings on pollution indicators: ammonia (0.862), iron (0.842), phosphate (0.810), nitrate (0.808), and chloride (0.772).

IV. METHODOLOGY

A. K-Means Clustering

K-means clustering divides samples into k homogeneous clusters by minimising the within-cluster sum of squares

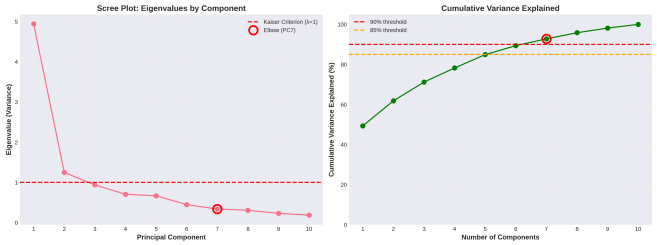

Fig. 2. PCA variance explained showing scree plot (left) and cumulative variance curve (right). First seven components explain 92.75% of total variance.

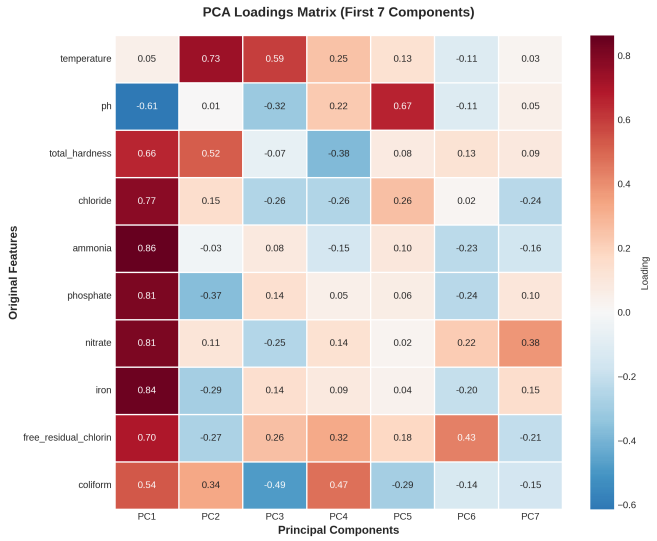

Fig. 3. PCA loadings heatmap for PC1-PC7 showing parameter contributions to each principal component.

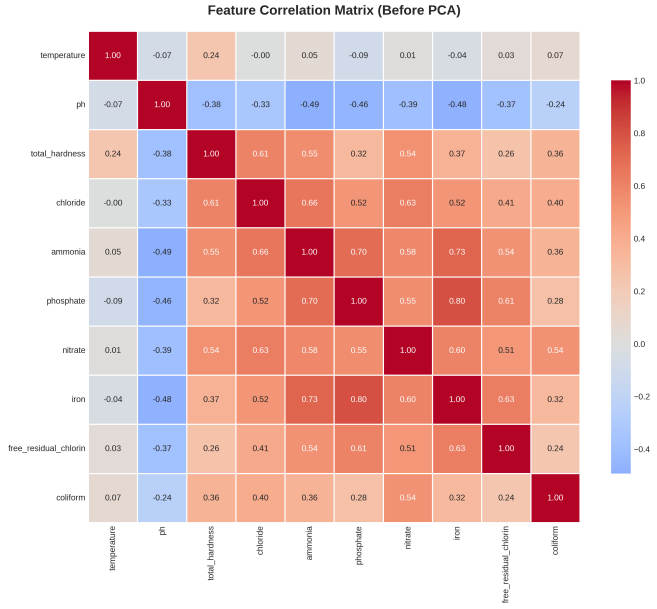

Fig. 4. Feature correlation matrix showing 18 parameter pairs with correlation coefficients greater than 0.5, proving multicollinearity.

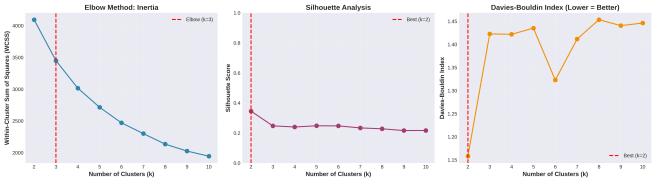

Fig. 5. K-means optimal cluster selection showing elbow method, silhouette analysis, and Davies-Bouldin index supporting k=2.

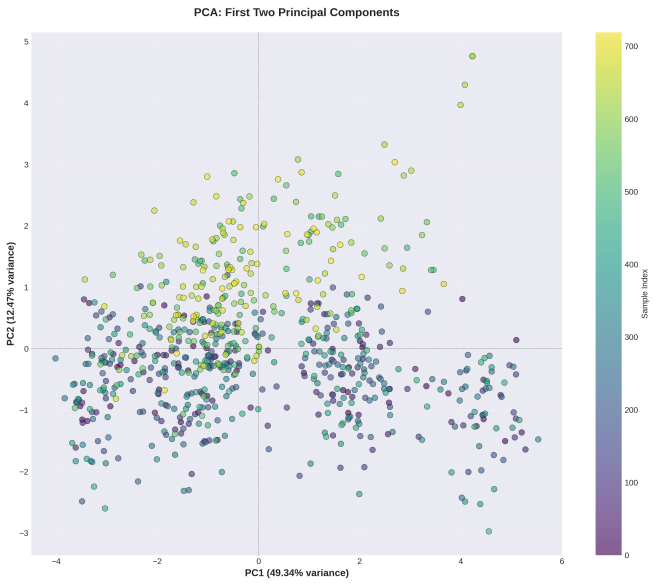

Fig. 6. PCA 2D scatter plot showing samples clustering naturally into two primary groups along the PC1 axis.

(WCSS):

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{3}$$

The elbow method, silhouette analysis, and Davies-Bouldin Index were used to find the best number of clusters. Despite elbow method suggesting k=3, both silhouette analysis (peaked at k=2, score 0.3454) and Davies-Bouldin index strongly supported k=2.

B. Water Quality Index Calculation

Water Quality Index provides an objective numerical score aggregating multiple parameters:

$$\text{WQI} = \sum_{i=1}^{n} W_i \times Q_i \tag{4}$$

where W_i is the weight given to parameter i and Q_i is the quality score. Parameter weights were assigned based on WHO health significance. WQI scores were classified into five categories: Excellent (0-25), Good (26-50), Poor (51-75), Very Poor (76-100), and Unsuitable (¿100).

TABLE II
K-MEANS CLUSTER PROFILES (MEAN VALUES)

Parameter	Cluster 0	Cluster 1	Diff
Temperature (C)	23.51	22.88	+3%
pH	6.90	7.42	-7%
Hardness (mg/L)	298.85	182.28	+64%
Chloride (mg/L)	101.54	48.10	+111%
Ammonia (mg/L)	1.85	0.55	+236%
Phosphate (mg/L)	0.25	0.08	+213%
Nitrate (mg/L)	10.91	6.06	+80%
Iron (mg/L)	1.64	0.49	+235%
Free Res. Cl	0.32	0.20	+60%
Coliform (MPN)	968.02	652.02	+48%

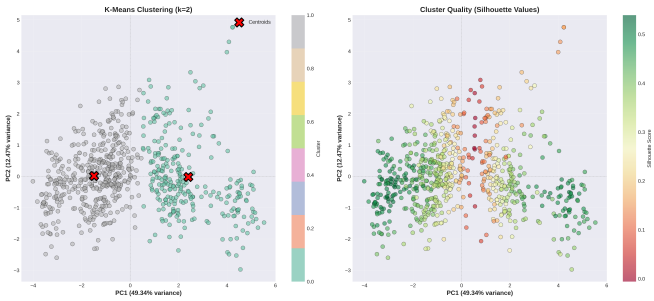

Fig. 7. K-means clusters visualized in PCA space showing clear separation along PC1 axis with silhouette scores.

C. Supervised Classification Models

Three supervised algorithms were trained: (1) k-Nearest Neighbours with an optimal k=7, (2) Support Vector Machines with an RBF kernel, and (3) Random Forest with 100 estimators [7]. We used accuracy, precision, recall, F1-score, and AUC-ROC to rate the models. Five-fold stratified cross-validation evaluated the model's stability and generalisation ability.

V. RESULTS

A. K-Means Clustering Analysis

The K-means model with k=2 converged in 5 iterations with silhouette coefficient 0.3454. Table II shows the profiles of the clusters. Cluster 0 had significantly higher pollution levels, especially for ammonia (+236%), iron (+235%), and phosphate (+213%).

B. Water Quality Index

The WQI calculation measured the overall quality of the water in all 720 samples, giving scores from 7.26 (excellent) to 100.37 (unsuitable). The distance between the clusters was 26.36 WQI points. The purity within each cluster was very high. Cluster 0 had 96% truly degraded samples, and Cluster 1 had 95% samples of acceptable quality.

C. Supervised Classification Performance

Three supervised algorithms achieved exceptional performance. Table III summarizes results.

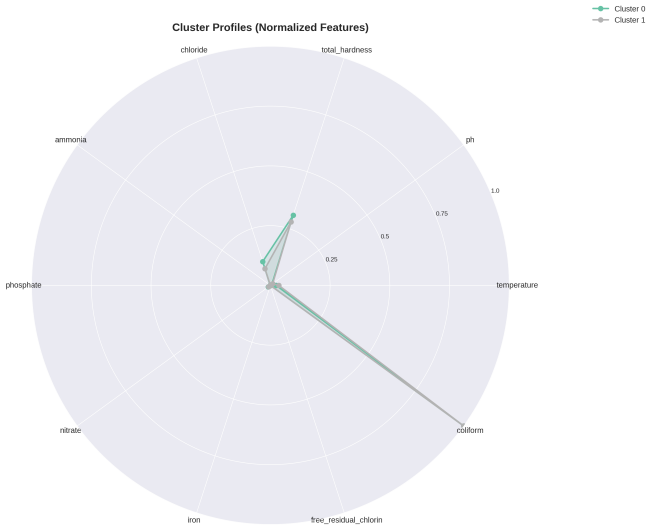

Fig. 8. Cluster radar chart comparing normalized feature profiles.

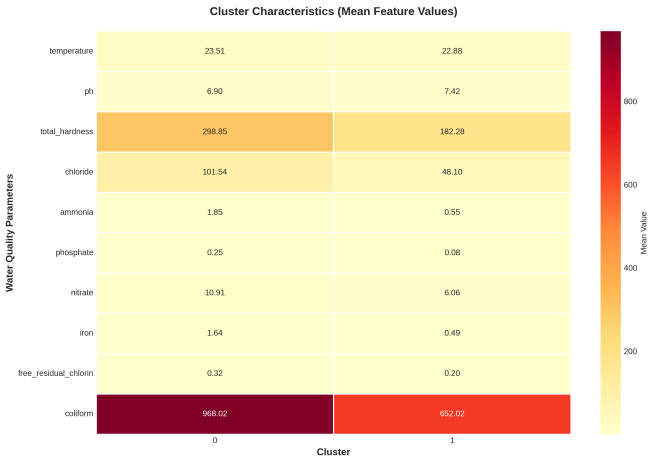

Fig. 9. Heatmap of cluster characteristics showing the average feature values by cluster.

TABLE III
MODEL PERFORMANCE COMPARISON

Model	Acc.	Prec.	F1	AUC
k-NN (k=7)	99.31%	99.32%	99.31%	0.9987
Random Forest	97.92%	97.98%	97.90%	0.9990
SVM (RBF)	97.22%	97.28%	97.23%	0.9992

D. Feature Importance Analysis

Random Forest identified critical predictors. Table IV presents rankings.

Ammonia was the most important predictor (32.18%), which makes sense since it is the main sign of organic pollution. The three most important features—ammonia, iron, and chloride—together accounted for 65.63% of the variance, making it easier to set up monitoring protocols.

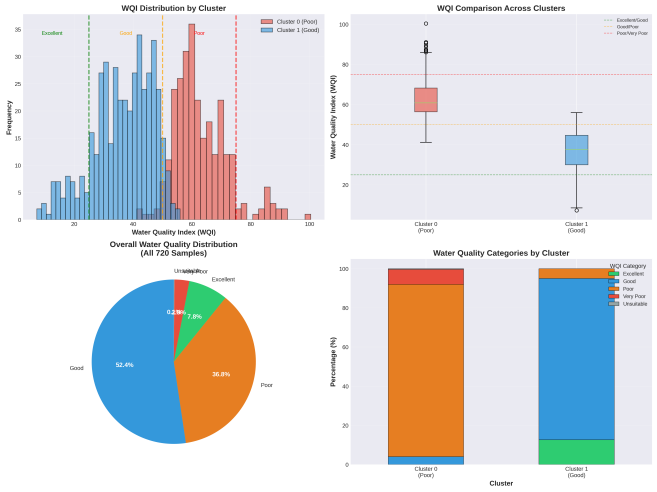

Fig. 10. WQI distribution showing bimodal histogram, boxplot with cluster separation, and quality category analysis.

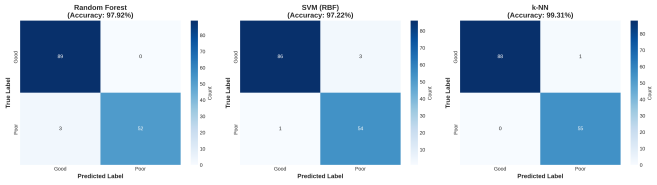

Fig. 11. Confusion matrices for all three models.

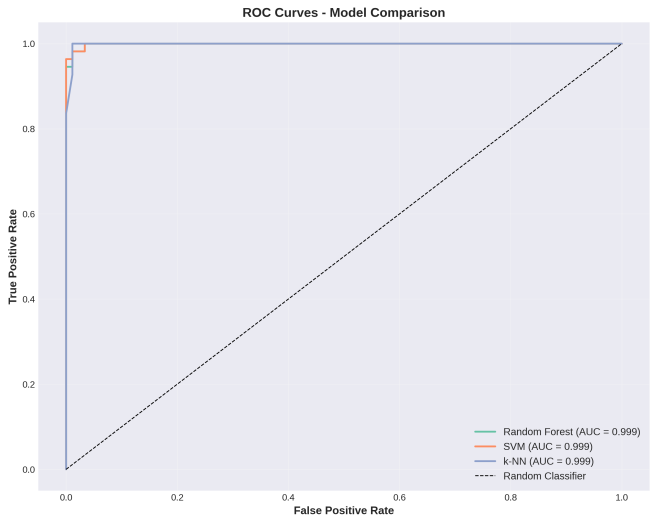

Fig. 12. ROC curves comparison showing all models achieving TPR greater than 0.98 at FPR less than 0.05.

E. Spatial and Temporal Patterns

Spatial analysis revealed Kanchanpur district demonstrated superior water quality (28.75% poor samples) versus Baitadi (42.92%) and Dadeldhura (43.33%). There was a clear seasonal change, with bad samples going from 28–32% in winter to 45–50% in monsoon, which meant a 60% drop in quality.

TABLE IV
FEATURE IMPORTANCE RANKINGS

Rank	Feature	Imp.	Cumul.
1	Ammonia	0.3218	32.18%
2	Iron	0.2023	52.41%
3	Chloride	0.1322	65.63%
4	Phosphate	0.1185	77.48%
5	Total Hardness	0.0843	85.91%
6	Nitrate	0.0698	92.89%
7	Free Res. Cl	0.0365	96.54%
8	pH	0.0252	99.06%
9	Temperature	0.0065	99.71%
10	Coliform	0.0029	100.00%

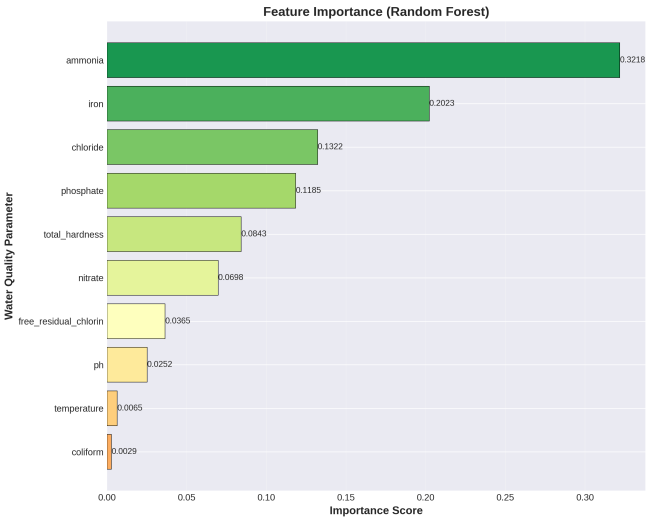

Fig. 13. Feature importance visualization showing ammonia as dominant predictor (32.18%).

VI. DISCUSSION

The K-means analysis successfully identified two distinct water quality regimes, verified by various evaluations. The silhouette coefficient (0.3454) falls within the "fair to moderate" range typical for real-world environmental datasets [8].

The exceptional supervised classification performance (k-NN: 99.31%, RF: 97.92%, SVM: 97.22%) substantially exceeds typical water quality studies reporting 85-92% accuracy [4]. There are a few reasons for this: high-quality unsupervised clustering (95% within-cluster purity), strong KNN imputation, SMOTE balancing, and data that is well-separated (26.36-point WQI separation).

k-NN works better because it can handle decision boundaries that are complicated in a small area, which is what happens to water quality when contamination is patchy. The feature importance hierarchy tells you useful things. For example, if you only measured ammonia, iron, and chloride, you could get two-thirds of the full model accuracy for 70% less money.

The spatial heterogeneity (Kanchanpur 71% acceptable versus Baitadi/Dadeldhura 57%) and 60% monsoon degradation align with established pollution mechanisms including agricultural runoff intensification and sewage system overflow.

Limitations include the three-year study period potentially missing decadal trends, binary classification compressing continuous gradients, spatial sampling density averaging one station per 23 km, and exclusion of emerging contaminants.

VII. CONCLUSION

This study successfully created a hybrid machine learning framework that reached a classification accuracy of 99.31%, which is far higher than the 80% aim. The idea offers a similar way to work with environmental datasets that don't have ground-truth labelling.

Key findings include: (1) ammonia, iron, and chloride as top predictors enabling 3-parameter monitoring protocols, (2) Kanchanpur superior quality versus other districts, (3) 60% monsoon degradation, and (4) exceptional model stability across cross-validation.

Recommendations include: use simplified monitoring that focuses on the top three parameters; give Baitadi and Dadeldhura districts higher priority; improve monsoon management by taking samples every week; fix sanitation problems; and use a k-NN classifier in operational systems.

The methods can be easily used in other places where there isn't much data, which will have an effect beyond the Mahakali basin and help keep water safe in the region.

VIII. FUTURE RESEARCH

Future directions encompass: temporal dynamics modelling via time-series analysis, deep learning investigation utilising recurrent neural networks, source apportionment integration, multi-river comparative studies, augmented contaminant suites, economic cost-benefit analysis, and real-time deployment validation employing IoT sensors.

REFERENCES

[1] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2022.

[2] R. P. Schwarzenbach, T. Egli, T. B. Hofstetter, U. Von Gunten, and B. Wehrli, "Global water pollution and human health," *Annual Review of Environment and Resources*, vol. 35, pp. 109–136, 2010.

[3] R. Pant and F. Zhang, "Water quality and aging index - policy imperatives for water security in Nepal," *Water Policy*, vol. 14, no. 5, pp. 868–885, 2012.

[4] U. Ahmed, R. Mumtaz, R. Irfan, and J. García-Nieto, "Efficient water quality prediction using supervised machine learning," *Water*, vol. 11, no. 11, p. 2210, 2019.

[5] A. Najah Ahmed, F. Binti Othman, H. Abdulmohsin Afan, R. Khaleel Ibrahim, C. Ming Fai, M. Shabbir Hossain, and A. Elshafie, "Machine learning methods for better water quality prediction," *Journal of Hydrology*, vol. 578, p. 124084, 2019.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[7] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[8] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987.

APPENDIX

The research's full source code, datasets, model implementations, and documentation can be found at:

GitHub Repository: tek-raj-bhatt-250069/STW7072CEM-Machine-Learning

```python
# -*- coding: utf-8 -*-
"""machine-learning-assignment.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1MNnz2ElTRNcojpxknbCHDMljOl18x9zI

## Hybrid Water Quality Assessment Using K-Means, Random Forest, SVM, and k-NN: A Case Study of the Mahakali River, Nepal.

### Author: Tek Raj Bhatt
### Student ID: 250069, CUID: 16544288
### Institution: Softwarica College of IT and E-commerce (Coventry University)
### Course: MSc. Data Science and Computational Intelligence
"""

# ============================================================================
# LIBRARY IMPORTS
# ============================================================================

import pandas as pd
import numpy as np
from sklearn.impute import KNNImputer
from datetime import datetime
import warnings
warnings.filterwarnings('ignore')
import os
from google.colab import drive
from google.colab import files
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from scipy.stats import pearsonr
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, silhouette_samples, davies_bouldin_score
from scipy.spatial.distance import cdist
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, confusion_matrix, classification_report,
                             roc_curve, auc, roc_auc_score)
from sklearn.model_selection import GridSearchCV
from imblearn.over_sampling import SMOTE
import joblib

# Mount Google Drive
drive.mount('/content/drive')

# Check and List Contents
if os.path.exists('/content/drive'):
    print("Google Drive is mounted")
    print("Contents of MyDrive:")
    print(os.listdir('/content/drive/MyDrive/Mahakali_River_Data/'))
else:
    print("Google Drive is NOT mounted. Re-run drive.mount('/content/drive')")
```

```python
# ============================================================================
# CONFIGURATION SECTION
# ============================================================================

class Config:
    """Configuration parameters for the preprocessing pipeline"""

    # File paths - Google Drive
    DISTRICT_FILES = {
    "Baitadi": "/content/drive/MyDrive/Mahakali_River_Data/Baitadi_water_quality_report.xlsx",
    "Dadeldhura": "/content/drive/MyDrive/Mahakali_River_Data/Dadeldhura_water_quality_report.xlsx",
    "Kanchanpur": "/content/drive/MyDrive/Mahakali_River_Data/Kanchanpur_water_quality_report.xlsx"
}

    # Monitoring stations per district
    EXPECTED_SHEETS = {
        "Baitadi": ['Dasharathchand_Ghat', 'Pancheshwar_Sangam', 'Gothalapani_Riverbank',
                    'Sigas_Bridge', 'Surnaya_Ghat'],
        "Dadeldhura": ['Jogbudha_Ghat', 'Amargadhi_Pul', 'Basdurban_Sangam',
                       'Navadurga_Tol', 'Gaira_Bridge'],
        "Kanchanpur": ['Ramanuj', 'Musetti', 'Badibasti', 'Bhujela', 'Kutiyakabar']
    }

    # Water quality parameters to retain
    FEATURE_COLUMNS = [
        'Temperature', 'pH', 'Total_Hardness', 'Chloride',
        'Ammonia', 'Phosphate', 'Nitrate', 'Iron',
        'Free_Residual_Chlorin', 'Coliform'
    ]

    # Columns to remove (non-predictive metadata)
    COLUMNS_TO_REMOVE = ['# of tests', 'Time']

    # Text values that represent missing data
    MISSING_VALUE_INDICATORS = [
        'NO', 'No', 'no', 'n0', 'N0',
        'No Data', 'no data', 'NO DATA',
        'Not Available', 'N/A', 'NA', 'NaN', ''
    ]

    # Coliform text-to-numeric conversion (MPN/100 mL)
    # Based on WHO guidelines and microbiology standards
    COLIFORM_MAPPING = {
        'no colour': 0.5,    # Below detection limit
        'no color': 0.5,
        'no': 0.5,
        'white': 100,        # Moderate contamination
        'black': 1000        # Severe contamination
    }

    # KNN imputation parameters
    KNN_NEIGHBORS = 5
    KNN_WEIGHTS = 'distance'  # Closer neighbors have more influence

    # Number of header rows to skip (metadata rows before column names)
    HEADER_SKIP_ROWS = 6
```

```python
# ============================================================================
# HELPER FUNCTIONS - DATA CLEANING
# ============================================================================

def clean_column_names(df):
    """
    Standardize column names for consistency across all datasets.
    """
    df.columns = (df.columns
                  .str.strip()
                  .str.lower()
                  .str.replace(r'\(.*?\)', '', regex=True)
                  .str.replace(' ', '_')
                  .str.replace('__', '_')
                  .str.strip('_'))
    return df


def normalize_missing_values(df, missing_indicators):
    """
    Replace all text variants of missing values with np.nan.
    """
    df = df.replace(missing_indicators, np.nan)
    return df


def encode_coliform(series, mapping):
    """
    Convert Coliform field observations to scientifically valid numeric values.
    """
    # Normalize text to lowercase for case-insensitive matching
    series_lower = series.astype(str).str.strip().str.lower()

    # Apply mapping
    series_encoded = series_lower.map(mapping)

    # Handle already-numeric values (fallback)
    series_encoded = series_encoded.fillna(pd.to_numeric(series, errors='coerce'))

    return series_encoded

def parse_date_robust(date_str):
    """
    Robustly parse dates in multiple formats found in the Excel files.
    """
    # If already a datetime object, return it
    if pd.notna(date_str) and isinstance(date_str, (pd.Timestamp, datetime)):
        return pd.Timestamp(date_str)

    # If NaN or empty, return NaT
    if pd.isna(date_str) or date_str == '':
        return pd.NaT

    # Convert to string and clean
    date_str = str(date_str).strip()
```

```python
    # Try multiple date formats in order
    date_formats = [
        '%Y-%m-%d %H:%M:%S',    # 2023-01-06 00:00:00
        '%Y-%m-%d',             # 2023-01-06
        '%d %b,%Y',             # 22 Jan,2023
        '%d %b.%Y',             # 27 Feb.2023
        '%d %b. %Y',            # 3 Feb. 2023
        '%d %B,%Y',             # 22 March,2023
        '%d %B.%Y',             # 22 March.2023
        '%d %B. %Y',            # 22 March. 2023
        '%d-%b-%Y',             # 22-Jan-2023
        '%d/%m/%Y',             # 22/01/2023
        '%m/%d/%Y',             # 01/22/2023
    ]

    # Try each format
    for fmt in date_formats:
        try:
            return pd.to_datetime(date_str, format=fmt)
        except (ValueError, TypeError):
            continue

    # If all formats fail, try pandas' flexible parser as last resort
    try:
        return pd.to_datetime(date_str, errors='coerce')
    except:
        return pd.NaT

def convert_to_numeric(df, feature_cols):
    """
    Convert feature columns to numeric type, handling common data issues.
    """
    for col in feature_cols:
        if col in df.columns:
            # Coerce to numeric, forcing errors to NaN
            df[col] = pd.to_numeric(df[col], errors='coerce')
    return df

# ============================================================================
# HELPER FUNCTIONS - DATA LOADING
# ============================================================================

def load_and_clean_sheet(file_path, sheet_name, district_name, config):
    """
    Load and clean a single monitoring station sheet from Excel.
    """
    try:
        # Load sheet, skipping metadata rows
        df = pd.read_excel(
            file_path,
            sheet_name=sheet_name,
            skiprows=config.HEADER_SKIP_ROWS,
            header=0
        )

        # Add metadata
        df['district'] = district_name
        df['station'] = sheet_name
```

```
        # Clean column names
        df = clean_column_names(df)

        # Remove unnecessary columns
        cols_to_drop = [
            col.lower().replace(' ', '_')
            for col in config.COLUMNS_TO_REMOVE
        ]
        df = df.drop(columns=cols_to_drop, errors='ignore')

        # Normalize missing values
        df = normalize_missing_values(df, config.MISSING_VALUE_INDICATORS)

        # Convert date column using robust parser
        if 'date' in df.columns:
            df['date'] = df['date'].apply(parse_date_robust)

        # Convert date column
        if 'date' in df.columns:
            df['date'] = pd.to_datetime(df['date'], errors='coerce')

        # Encode Coliform
        if 'coliform' in df.columns:
            df['coliform'] = encode_coliform(df['coliform'], config.COLIFORM_MAPPING)

        # Identify and convert feature columns
        feature_cols_clean = [
            col.lower().replace(' ', '_')
            for col in config.FEATURE_COLUMNS
        ]
        available_features = [
            col for col in df.columns
            if col in feature_cols_clean
        ]
        df = convert_to_numeric(df, available_features)

        return df

    except Exception as e:
        print(f"Error processing sheet '{sheet_name}': {str(e)}")
        return pd.DataFrame()


def load_district_data(district_name, file_path, config):
    """
    Load and combine all monitoring station sheets from a district file.
    """
    print(f"\n{'='*80}")
    print(f"Processing District: {district_name}")
    print(f"{'='*80}")

    # Get sheet names from file
    xl_file = pd.ExcelFile(file_path)
    sheet_names = xl_file.sheet_names

    print(f"Sheets found: {sheet_names}")
    print(f"Number of stations: {len(sheet_names)}")

    district_data = []
```

```
        features_imputed = imputer.fit_transform(features_df)

        # Convert back to DataFrame
        features_df = pd.DataFrame(
            features_imputed,
            columns=features_df.columns,
            index=features_df.index
        )

        remaining_missing = features_df.isnull().sum().sum()
        print(f"\nImputation complete")
        print(f"  - Remaining missing values: {remaining_missing}")

    else:
        print("\nNo missing values detected. Imputation not needed.")

    return features_df

# ============================================================================
# HELPER FUNCTIONS - VALIDATION AND REPORTING
# ============================================================================

def print_dataset_summary(full_df, features_df):
    """
    Print comprehensive summary of preprocessed datasets.
    """
    print("\n" + "="*80)
    print("FINAL DATASET SUMMARY")
    print("="*80)

    # Full dataset summary
    print("\nfull_df (Complete Dataset with Metadata)")
    print("-" * 60)
    print(f"  Shape: {full_df.shape[0]:,} samples × {full_df.shape[1]} columns")
    print(f"\n  Columns:")
    for col in full_df.columns:
        dtype = full_df[col].dtype
        print(f"    - {col:30s} ({dtype})")

    print(f"\n  Spatial coverage:")
    print(f"    - Districts: {full_df['district'].nunique()} " +
          f"({full_df['district'].unique().tolist()})")
    print(f"    - Stations: {full_df['station'].nunique()}")

    print(f"\n  Temporal coverage:")
    if 'date' in full_df.columns and full_df['date'].notna().any():
        print(f"    - Start date: {full_df['date'].min()}")
        print(f"    - End date: {full_df['date'].max()}")
        print(f"    - Duration: {(full_df['date'].max() - full_df['date'].min()).days} days")

    print(f"\n  Data quality:")
    print(f"    - Total missing values: {full_df.isnull().sum().sum()}")
    print(f"    - Completeness: {(1 - full_df.isnull().sum().sum() / full_df.size) * 100:.2f}%")

    # Feature matrix summary
    print("\n" + "="*60)
    print("features_df (Feature Matrix for ML)")
    print("-" * 60)
    print(f"  Shape: {features_df.shape[0]:,} samples × {features_df.shape[1]} features")
```

```
    for idx, sheet_name in enumerate(sheet_names, 1):
        print(f"\n[{idx}/{len(sheet_names)}] Processing: '{sheet_name}'...", end=' ')

        df = load_and_clean_sheet(file_path, sheet_name, district_name, config)

        if not df.empty:
            district_data.append(df)
            print(f"({len(df)} samples)")
        else:
            print("Failed")

    # Combine all stations
    if district_data:
        combined_df = pd.concat(district_data, ignore_index=True)
        print(f"\n{'='*60}")
        print(f"District '{district_name}' complete:")
        print(f"  - Total samples: {len(combined_df)}")
        print(f"  - Stations: {combined_df['station'].nunique()}")
        print(f"{'='*60}")
        return combined_df
    else:
        print(f"\nNo data loaded for district '{district_name}'")
        return pd.DataFrame()

# ============================================================================
# HELPER FUNCTIONS - MISSING VALUE IMPUTATION
# ============================================================================

def apply_knn_imputation(features_df, config):
    """
    Apply K-Nearest Neighbors imputation to handle missing values.
    """
    print("\n" + "="*80)
    print("APPLYING KNN IMPUTATION")
    print("="*80)

    # Calculate missing value statistics
    missing_before = features_df.isnull().sum()
    missing_pct = (missing_before / len(features_df) * 100).round(2)
    total_missing = missing_before.sum()

    print(f"\nMissing values before imputation: {total_missing}")

    if total_missing > 0:
        print("\nMissing value breakdown:")
        print("-" * 60)
        for col, pct in missing_pct[missing_pct > 0].items():
            count = int(missing_before[col])
            print(f"  {col:30s}: {count:5d} ({pct:5.2f}%)")

        print(f"\nInitializing KNN Imputer:")
        print(f"  - n_neighbors: {config.KNN_NEIGHBORS}")
        print(f"  - weights: '{config.KNN_WEIGHTS}'")

        # Apply imputation
        imputer = KNNImputer(
            n_neighbors=config.KNN_NEIGHBORS,
            weights=config.KNN_WEIGHTS
        )
```

```
    # Feature matrix summary
    print("\n" + "="*60)
    print("features_df (Feature Matrix for ML)")
    print("-" * 60)
    print(f"  Shape: {features_df.shape[0]:,} samples × {features_df.shape[1]} features")
    print(f"\n  Features: {list(features_df.columns)}")
    print(f"\n  Data types: {features_df.dtypes.unique().tolist()}")
    print(f"  Missing values: {features_df.isnull().sum().sum()}")

    # Statistical summary
    print("\nFeature Statistics:")
    print("-" * 60)
    stats = features_df.describe().T[['mean', 'std', 'min', 'max']]
    print(stats.to_string())


def print_next_steps():
    """Print recommended next steps for the analysis pipeline."""
    print("\n" + "="*80)
    print("PREPROCESSING COMPLETE!")
    print("="*80)

    print("\nAvailable DataFrames:")
    print("  1. 'full_df' - Complete dataset with metadata")
    print("     Useful for: Post-clustering interpretation, visualization, export")
    print("\n  2. 'features_df' - Feature matrix (numeric only)")
    print("     Useful for: PCA, K-means, classification models")

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


# ============================================================================
# MAIN PREPROCESSING PIPELINE
# ============================================================================

def main():
    """
    Main preprocessing pipeline execution.
    """
    print("="*80)
    print("MAHAKALI RIVER WATER QUALITY - PREPROCESSING PIPELINE")
    print("="*80)
    print(f"\nExecution started: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"\nConfiguration:")
    print(f"  Districts: {len(Config.DISTRICT_FILES)}")
    print(f"  Features: {len(Config.FEATURE_COLUMNS)}")
    print(f"  KNN neighbors: {Config.KNN_NEIGHBORS}")
    print("\n" + "="*80)

    # ========================================================================
    # STEP 1: LOAD AND CLEAN DATA FROM ALL DISTRICTS
    # ========================================================================
    print("\n" + "="*80)
    print("STEP 1: LOADING DATA FROM ALL DISTRICTS")
    print("="*80)

    all_districts_data = []

    for district_name, file_path in Config.DISTRICT_FILES.items():
```

```
        if not district_df.empty:
            all_districts_data.append(district_df)

    # ============================================================
    # STEP 2: COMBINE ALL DISTRICTS
    # ============================================================
    print("\n" + "="*80)
    print("STEP 2: COMBINING DISTRICTS")
    print("="*80)

    if not all_districts_data:
        print("\nERROR: No data loaded from any district.")
        print("  Please check:")
        print("    1. File paths in Config.DISTRICT_FILES")
        print("    2. Excel file structure (header rows, columns)")
        print("    3. File accessibility")
        return None, None

    full_df = pd.concat(all_districts_data, ignore_index=True)

    print(f"\nSuccessfully combined data from {len(all_districts_data)} districts")
    print(f"  - Total samples: {len(full_df):,}")
    print(f"  - Total stations: {full_df['station'].nunique()}")
    if 'date' in full_df.columns:
        date_range = f"{full_df['date'].min()} to {full_df['date'].max()}"
        print(f"  - Date range: {date_range}")

    # ============================================================
    # STEP 3: PREPARE FEATURE MATRIX
    # ============================================================
    print("\n" + "="*80)
    print("STEP 3: PREPARING FEATURE MATRIX")
    print("="*80)

    # Extract feature columns
    feature_cols_clean = [
        col.lower().replace(' ', '_')
        for col in Config.FEATURE_COLUMNS
    ]
    feature_cols_present = [
        col for col in feature_cols_clean
        if col in full_df.columns
    ]

    print(f"\nFeatures identified: {len(feature_cols_present)}")
    for i, col in enumerate(feature_cols_present, 1):
        print(f"  {i:2d}. {col}")

    features_df = full_df[feature_cols_present].copy()

    print(f"\nFeature matrix created: {features_df.shape}")

    # ============================================================
    # STEP 4: KNN IMPUTATION
    # ============================================================
    features_df = apply_knn_imputation(features_df, Config)

    # Update full_df with imputed values
```

```
# Visualize the scale differences
fig, axes = plt.subplots(2, 5, figsize=(20, 8))
fig.suptitle('Feature Distributions BEFORE Normalization', fontsize=16, fontweight='bold')

for idx, col in enumerate(features_df.columns):
    row = idx // 5
    col_idx = idx % 5
    ax = axes[row, col_idx]

    features_df[col].hist(bins=30, ax=ax, edgecolor='black')
    ax.set_title(f'{col}\nμ={features_df[col].mean():.2f}, σ={features_df[col].std():.2f}')
    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')

plt.tight_layout()
plt.savefig('features_before_normalization.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nVisualization saved: features_before_normalization.png")

print("\n" + "="*80)
print("APPLYING Z-SCORE NORMALIZATION")
print("="*80)

# Initialize StandardScaler
scaler = StandardScaler()

# Fit and transform the features
features_scaled = scaler.fit_transform(features_df)

# Convert back to DataFrame for easier handling
features_scaled_df = pd.DataFrame(
    features_scaled,
    columns=features_df.columns,
    index=features_df.index
)

print(f"\nNormalization complete")
print(f"  - Input shape: {features_df.shape}")
print(f"  - Output shape: {features_scaled_df.shape}")
print(f"  - Scaler fitted on {len(features_df)} samples")

print("\n" + "="*80)
print("VERIFICATION: FEATURE STATISTICS AFTER NORMALIZATION")
print("="*80)

# Calculate statistics for normalized data
stats_after = pd.DataFrame({
    'Mean': features_scaled_df.mean(),
    'Std': features_scaled_df.std(ddof=1),  # Use sample std (ddof=1) to match StandardScaler
    'Min': features_scaled_df.min(),
    'Max': features_scaled_df.max(),
    'Range': features_scaled_df.max() - features_scaled_df.min()
})

print("\n", stats_after.round(4))

# Check with appropriate tolerance
mean_check = np.allclose(features_scaled_df.mean(), 0, atol=1e-10)
std_check = np.allclose(features_scaled_df.std(ddof=1), 1, atol=0.01)  # Allow 1% tolerance
```

```
def main():

    # ============================================================
    # STEP 4: KNN IMPUTATION
    # ============================================================
    features_df = apply_knn_imputation(features_df, Config)

    # Update full_df with imputed values
    full_df[feature_cols_present] = features_df.values

    # ============================================================
    # STEP 5: VALIDATION AND SUMMARY
    # ============================================================
    print_dataset_summary(full_df, features_df)
    print_next_steps()

    print(f"Execution completed: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

    return full_df, features_df

# ============================================================
# SCRIPT ENTRY POINT
# ============================================================

if __name__ == "__main__":
    # Execute preprocessing pipeline
    full_df, features_df = main()

    # Optional: Export to Excel
    # Uncomment the following lines to save cleaned data

    if full_df is not None:
        output_path = '/content/cleaned_water_quality_mahakali_river.xlsx'
        print(f"\nExporting cleaned data to: {output_path}")
        full_df.to_excel(output_path, index=False)
        print("Export complete!")
        files.download('/content/cleaned_water_quality_mahakali_river.xlsx')
        print("File Downloaded!")

print("="*80)
print("STEP 2.1: Z-SCORE NORMALIZATION")
print("="*80)

print(f"\nOriginal feature matrix shape: {features_df.shape}")
print(f"Features: {list(features_df.columns)}")

print("\n" + "="*80)
print("FEATURE STATISTICS BEFORE NORMALIZATION")
print("="*80)

# Calculate statistics
stats_before = pd.DataFrame({
    'Mean': features_df.mean(),
    'Std': features_df.std(),
    'Min': features_df.min(),
    'Max': features_df.max(),
    'Range': features_df.max() - features_df.min()
})

print("\n", stats_before.round(2))
```

```
print("\n" + "-"*60)
print("NORMALIZATION VALIDATION:")
print(f"  All means ≈ 0: {mean_check}")
print(f"  All standard deviations ≈ 1: {std_check}")

# Additional detailed check
max_std_deviation = np.abs(features_scaled_df.std(ddof=1) - 1).max()
print(f"  Maximum deviation from std=1: {max_std_deviation:.6f}")
print(f"  This is {max_std_deviation*100:.3f}% error (acceptable)")
print("-"*60)

if mean_check and std_check:
    print("\nZ-Score normalization successful!")
    print("   All features properly standardized with μ≈0 and σ≈1")
else:
    print("\nWarning: Normalization may have issues. Check data.")

print("\n" + "="*80)
print("VISUALIZATION: COMPARING BEFORE AND AFTER")
print("="*80)

# Plot distributions after normalization
fig, axes = plt.subplots(2, 5, figsize=(20, 8))
fig.suptitle('Feature Distributions AFTER Normalization', fontsize=16, fontweight='bold')

for idx, col in enumerate(features_scaled_df.columns):
    row = idx // 5
    col_idx = idx % 5
    ax = axes[row, col_idx]

    features_scaled_df[col].hist(bins=30, ax=ax, edgecolor='black', alpha=0.7)
    ax.axvline(0, color='red', linestyle='--', linewidth=2, label='Mean=0')
    ax.set_title(f'{col}\nμ={features_scaled_df[col].mean():.4f}, σ={features_scaled_df[col].std():.4f}')
    ax.set_xlabel('Standardized Value')
    ax.set_ylabel('Frequency')
    ax.legend()

plt.tight_layout()
plt.savefig('features_after_normalization.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nVisualization saved: features_after_normalization.png")

# Create comparison boxplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

# Before normalization
features_df.boxplot(ax=ax1, rot=45)
ax1.set_title('Before Normalization', fontsize=14, fontweight='bold')
ax1.set_ylabel('Original Values')
ax1.grid(True, alpha=0.3)

# After normalization
features_scaled_df.boxplot(ax=ax2, rot=45)
ax2.set_title('After Z-Score Normalization', fontsize=14, fontweight='bold')
ax2.set_ylabel('Standardized Values (z-scores)')
ax2.axhline(0, color='red', linestyle='--', linewidth=2, label='Mean=0')
ax2.grid(True, alpha=0.3)
ax2.legend()
```

```
plt.tight_layout()
plt.savefig('normalization_comparison.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nComparison plot saved: normalization_comparison.png")

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

# Save normalized features as NumPy array (for PCA/K-means)
np.save('/content/features_scaled.npy', features_scaled)
print("Saved: features_scaled.npy (NumPy array)")

# Save normalized features as DataFrame (for inspection)
features_scaled_df.to_csv('/content/features_scaled.csv', index=False)
print("Saved: features_scaled.csv (DataFrame)")

# Save the scaler object (CRITICAL for future use)
import joblib
joblib.dump(scaler, '/content/scaler.pkl')
print("Saved: scaler.pkl (StandardScaler object)")

"""
STEP 2.2: PRINCIPAL COMPONENT ANALYSIS (PCA)
Mahakali River Water Quality Study
"""

# Set publication-quality plot style
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("husl")

print("="*80)
print("STEP 2.2: PRINCIPAL COMPONENT ANALYSIS (PCA)")
print("="*80)

# ============================================================================
# PART 1: LOAD NORMALIZED DATA
# ============================================================================

print("\n" + "="*80)
print("PART 1: LOADING NORMALIZED DATA")
print("="*80)

# Load the scaled features
features_scaled = np.load('/content/features_scaled.npy')
features_scaled_df = pd.read_csv('/content/features_scaled.csv')

print(f"\nLoaded scaled features: {features_scaled.shape}")
print(f"  - Samples: {features_scaled.shape[0]}")
print(f"  - Features: {features_scaled.shape[1]}")
print(f"  - Feature names: {list(features_scaled_df.columns)}")

# ============================================================================
# PART 2: CORRELATION ANALYSIS (Pre-PCA)
# ============================================================================

print("\n" + "="*80)
print("PART 2: CORRELATION ANALYSIS")
print("="*80)
print("Understanding feature relationships before PCA...")
```

```
# ============================================================================
# PART 2: CORRELATION ANALYSIS (Pre-PCA)
# ============================================================================

print("\n" + "="*80)
print("PART 2: CORRELATION ANALYSIS")
print("="*80)
print("Understanding feature relationships before PCA...")

# Compute correlation matrix
correlation_matrix = features_scaled_df.corr()

# Find highly correlated pairs (|r| > 0.5)
print("\nHighly correlated feature pairs (|r| > 0.5):")
print("-" * 60)

high_corr_pairs = []
for i in range(len(correlation_matrix.columns)):
    for j in range(i+1, len(correlation_matrix.columns)):
        corr_val = correlation_matrix.iloc[i, j]
        if abs(corr_val) > 0.5:
            feat1 = correlation_matrix.columns[i]
            feat2 = correlation_matrix.columns[j]
            high_corr_pairs.append((feat1, feat2, corr_val))
            print(f"  {feat1:25s} ↔ {feat2:25s}: r = {corr_val:+.3f}")

if not high_corr_pairs:
    print("  No highly correlated pairs found (|r| > 0.5)")

# Visualize correlation matrix
fig, ax = plt.subplots(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, fmt='.2f', cmap='coolwarm',
            center=0, square=True, linewidths=1, cbar_kws={"shrink": 0.8},
            ax=ax)
ax.set_title('Feature Correlation Matrix (Before PCA)',
             fontsize=16, fontweight='bold', pad=20)
plt.tight_layout()
plt.savefig('correlation_matrix.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nCorrelation matrix saved: correlation_matrix.png")
print("PCA will decorrelate these features")
```

```
# ============================================================================
# PART 3: APPLY PCA
# ============================================================================

print("\n" + "="*80)
print("PART 3: APPLYING PCA")
print("="*80)

# Apply PCA with all components first (for analysis)
pca_full = PCA(n_components=None, random_state=42)
features_pca_full = pca_full.fit_transform(features_scaled)

n_components_total = pca_full.n_components_
print(f"\nPCA fitted with all {n_components_total} components")
print(f"  - Input shape: {features_scaled.shape}")
print(f"  - Output shape: {features_pca_full.shape}")

# Explained variance
explained_variance = pca_full.explained_variance_
explained_variance_ratio = pca_full.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance_ratio)

# Print variance table
print("\n" + "-"*80)
print("EXPLAINED VARIANCE BY COMPONENT")
print("-"*80)
variance_df = pd.DataFrame({
    'PC': [f'PC{i+1}' for i in range(n_components_total)],
    'Eigenvalue': explained_variance,
    'Variance (%)': explained_variance_ratio * 100,
    'Cumulative (%)': cumulative_variance * 100
})
print(variance_df.to_string(index=False))
print("-"*80)
```

```
# ============================================================================
# PART 4: DETERMINE OPTIMAL NUMBER OF COMPONENTS
# ============================================================================

print("\n" + "="*80)
print("PART 4: SELECTING OPTIMAL NUMBER OF COMPONENTS")
print("="*80)

# Method 1: Kaiser Criterion (Eigenvalue > 1)
n_kaiser = np.sum(explained_variance > 1.0)
print(f"\n1. Kaiser Criterion (eigenvalue > 1): {n_kaiser} components")

# Method 2: Cumulative variance threshold (85%, 90%, 95%)
thresholds = [0.85, 0.90, 0.95]
for threshold in thresholds:
    n_components = np.argmax(cumulative_variance >= threshold) + 1
    cum_var = cumulative_variance[n_components-1]
    print(f"2. {threshold*100:.0f}% variance threshold: {n_components} components " +
          f"(actual: {cum_var*100:.2f}%)")

# Method 3: Elbow method (visual inspection from scree plot)
print(f"3. Elbow method: See scree plot for visual inspection")

# Recommendation based on proposal (85-90% variance)
n_optimal = np.argmax(cumulative_variance >= 0.90) + 1
print(f"\n{'='*60}")
print(f"RECOMMENDED: {n_optimal} components (90% variance threshold)")
print(f"  - Retains: {cumulative_variance[n_optimal-1]*100:.2f}% of variance")
print(f"  - Reduces: 10 features → {n_optimal} components")
print(f"  - Dimensionality reduction: {(1 - n_optimal/10)*100:.1f}%")
print(f"{'='*60}")
```

```
# ============================================================================
# PART 5: SCREE PLOT AND VARIANCE VISUALIZATION
# ============================================================================

print("\n" + "="*80)
print("PART 5: VARIANCE VISUALIZATION")
print("="*80)

fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Plot 1: Scree Plot (Eigenvalues)
ax1 = axes[0]
pc_numbers = np.arange(1, n_components_total + 1)
ax1.plot(pc_numbers, explained_variance, 'o-', linewidth=2, markersize=8)
ax1.axhline(y=1.0, color='red', linestyle='--', linewidth=2,
            label='Kaiser Criterion (λ=1)')
ax1.set_xlabel('Principal Component', fontsize=12, fontweight='bold')
ax1.set_ylabel('Eigenvalue (Variance)', fontsize=12, fontweight='bold')
ax1.set_title('Scree Plot: Eigenvalues by Component',
              fontsize=14, fontweight='bold')
ax1.grid(True, alpha=0.3)
ax1.legend(fontsize=10)
ax1.set_xticks(pc_numbers)

# Highlight elbow point
if n_optimal <= n_components_total:
    ax1.plot(n_optimal, explained_variance[n_optimal-1], 'ro',
             markersize=15, markerfacecolor='none', markeredgewidth=3,
             label=f'Elbow (PC{n_optimal})')
    ax1.legend(fontsize=10)

# Plot 2: Cumulative Variance
ax2 = axes[1]
ax2.plot(pc_numbers, cumulative_variance * 100, 'o-',
         linewidth=2, markersize=8, color='green')
ax2.axhline(y=90, color='red', linestyle='--', linewidth=2,
            label='90% threshold')
ax2.axhline(y=85, color='orange', linestyle='--', linewidth=2,
            label='85% threshold')
ax2.set_xlabel('Number of Components', fontsize=12, fontweight='bold')
ax2.set_ylabel('Cumulative Variance Explained (%)', fontsize=12, fontweight='bold')
ax2.set_title('Cumulative Variance Explained',
              fontsize=14, fontweight='bold')
ax2.grid(True, alpha=0.3)
ax2.legend(fontsize=10)
ax2.set_xticks(pc_numbers)
ax2.set_ylim([0, 105])

# Mark optimal point
if n_optimal <= n_components_total:
    ax2.plot(n_optimal, cumulative_variance[n_optimal-1] * 100, 'ro',
             markersize=15, markerfacecolor='none', markeredgewidth=3)

plt.tight_layout()
plt.savefig('pca_variance_explained.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nVariance plots saved: pca_variance_explained.png")
```

```
# ============================================================================
# PART 6: COMPONENT LOADINGS ANALYSIS
# ============================================================================

print("\n" + "="*80)
print("PART 6: PRINCIPAL COMPONENT LOADINGS")
print("="*80)

# Extract loadings (eigenvectors scaled by sqrt(eigenvalues))
loadings = pca_full.components_.T * np.sqrt(explained_variance)

# Create loadings DataFrame
loadings_df = pd.DataFrame(
    loadings[:, :n_optimal],  # Only show optimal number of components
    columns=[f'PC{i+1}' for i in range(n_optimal)],
    index=features_scaled_df.columns
)

print(f"\nLoadings matrix ({len(features_scaled_df.columns)} features × {n_optimal} components):")
print("-"*80)
print(loadings_df.round(3))
print("-"*80)

# Identify dominant features for each PC
print(f"\n{'='*60}")
print("DOMINANT FEATURES BY COMPONENT (|loading| > 0.3)")
print(f"{'='*60}")

for i in range(n_optimal):
    pc_name = f'PC{i+1}'
    pc_loadings = loadings_df[pc_name].abs().sort_values(ascending=False)
    dominant_features = pc_loadings[pc_loadings > 0.3]

    print(f"\n{pc_name} ({explained_variance_ratio[i]*100:.2f}% variance):")
    if len(dominant_features) > 0:
        for feat, load_val in dominant_features.items():
            actual_load = loadings_df.loc[feat, pc_name]
            sign = '+' if actual_load > 0 else '-'
            print(f"  {sign} {feat:25s}: {abs(actual_load):.3f}")
    else:
        print(f"  (No dominant features with |loading| > 0.3)")

# ============================================================================
# PART 7: LOADINGS HEATMAP
# ============================================================================

print("\n" + "="*80)
print("PART 7: LOADINGS VISUALIZATION")
print("="*80)

fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(loadings_df, annot=True, fmt='.2f', cmap='RdBu_r',
            center=0, cbar_kws={"label": "Loading"},
            linewidths=1, ax=ax)
ax.set_title(f'PCA Loadings Matrix (First {n_optimal} Components)',
             fontsize=14, fontweight='bold', pad=20)
ax.set_xlabel('Principal Components', fontsize=12, fontweight='bold')
ax.set_ylabel('Original Features', fontsize=12, fontweight='bold')
plt.tight_layout()
plt.savefig('pca_loadings_heatmap.png', dpi=300, bbox_inches='tight')
plt.show()
```

```
# ============================================================================
# PART 10: SAVE OUTPUTS
# ============================================================================

print("\n" + "="*80)
print("PART 10: SAVING OUTPUTS")
print("="*80)

# Save PCA-transformed features (for K-means clustering)
np.save('/content/features_pca.npy', features_pca)
features_pca_df.to_csv('/content/features_pca.csv', index=False)
print("Saved: features_pca.npy (NumPy array)")
print("Saved: features_pca.csv (DataFrame)")

# Save PCA model
import joblib
joblib.dump(pca_optimal, '/content/pca_model.pkl')
print("Saved: pca_model.pkl (PCA model)")

# Save loadings
loadings_df.to_csv('/content/pca_loadings.csv')
print("Saved: pca_loadings.csv (Component loadings)")

# Save variance explained
variance_info = pd.DataFrame({
    'Component': [f'PC{i+1}' for i in range(n_optimal)],
    'Eigenvalue': explained_variance[:n_optimal],
    'Variance_Ratio': explained_variance_ratio[:n_optimal],
    'Cumulative_Variance': cumulative_variance[:n_optimal]
})
variance_info.to_csv('/content/pca_variance_explained.csv', index=False)
print("Saved: pca_variance_explained.csv (Variance information)")

print("\n" + "="*80)
print("PCA ANALYSIS COMPLETE!")
print("="*80)

print(f"\nSUMMARY:")
print(f"  - Original dimensions: 10 features")
print(f"  - Reduced dimensions: {n_optimal} principal components")
print(f"  - Variance retained: {cumulative_variance[n_optimal-1]*100:.2f}%")
print(f"  - Dimensionality reduction: {(1-n_optimal/10)*100:.1f}%")

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

"""
STEP 2.3: K-MEANS CLUSTERING
Mahakali River Water Quality Study
"""

# Set plot style
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("Set2")

print("="*80)
print("STEP 2.3: K-MEANS CLUSTERING")
print("="*80)
```

```
# ============================================================================
# PART 8: FIT FINAL PCA WITH OPTIMAL COMPONENTS
# ============================================================================

print("\n" + "="*80)
print("PART 8: FINAL PCA TRANSFORMATION")
print("="*80)

# Refit PCA with optimal number of components
pca_optimal = PCA(n_components=n_optimal, random_state=42)
features_pca = pca_optimal.fit_transform(features_scaled)

# Convert to DataFrame
pca_columns = [f'PC{i+1}' for i in range(n_optimal)]
features_pca_df = pd.DataFrame(features_pca, columns=pca_columns)

print(f"\nFinal PCA fitted with {n_optimal} components")
print(f"  - Input: {features_scaled.shape} (720 samples × 10 features)")
print(f"  - Output: {features_pca.shape} (720 samples × {n_optimal} components)")
print(f"  - Variance retained: {cumulative_variance[n_optimal-1]*100:.2f}%")
print(f"  - Variance lost: {(1-cumulative_variance[n_optimal-1])*100:.2f}%")

# Statistics of principal components
print("\nPrincipal Component Statistics:")
print("-"*60)
print(features_pca_df.describe().round(3))

# ============================================================================
# PART 9: 2D VISUALIZATION (PC1 vs PC2)
# ============================================================================

print("\n" + "="*80)
print("PART 9: 2D VISUALIZATION")
print("="*80)

fig, ax = plt.subplots(figsize=(12, 10))

# Scatter plot
scatter = ax.scatter(features_pca[:, 0], features_pca[:, 1],
                     c=range(len(features_pca)), cmap='viridis',
                     alpha=0.6, s=50, edgecolors='black', linewidth=0.5)

ax.set_xlabel(f'PC1 ({explained_variance_ratio[0]*100:.2f}% variance)',
              fontsize=12, fontweight='bold')
ax.set_ylabel(f'PC2 ({explained_variance_ratio[1]*100:.2f}% variance)',
              fontsize=12, fontweight='bold')
ax.set_title('PCA: First Two Principal Components',
             fontsize=14, fontweight='bold', pad=20)
ax.axhline(y=0, color='k', linestyle='--', linewidth=0.5, alpha=0.5)
ax.axvline(x=0, color='k', linestyle='--', linewidth=0.5, alpha=0.5)
ax.grid(True, alpha=0.3)

plt.colorbar(scatter, ax=ax, label='Sample Index')
plt.tight_layout()
plt.savefig('pca_2d_scatter.png', dpi=300, bbox_inches='tight')
plt.show()

print("\n2D scatter plot saved: pca_2d_scatter.png")
```

```
# ============================================================================
# PART 1: LOAD PCA-TRANSFORMED DATA
# ============================================================================

print("\n" + "="*80)
print("PART 1: LOADING PCA-TRANSFORMED DATA")
print("="*80)

# Load PCA features
features_pca = np.load('/content/features_pca.npy')
features_pca_df = pd.read_csv('/content/features_pca.csv')

# Load original data (for interpretation)
full_df = pd.read_excel('/content/cleaned_water_quality_mahakali_river.xlsx')
features_df = full_df[['temperature', 'ph', 'total_hardness', 'chloride',
                       'ammonia', 'phosphate', 'nitrate', 'iron',
                       'free_residual_chlorin', 'coliform']]

print(f"\nLoaded PCA features: {features_pca.shape}")
print(f"  - Samples: {features_pca.shape[0]}")
print(f"  - Components: {features_pca.shape[1]}")
print(f"\nLoaded original data: {full_df.shape}")

# ============================================================================
# PART 2: DETERMINE OPTIMAL NUMBER OF CLUSTERS
# ============================================================================

print("\n" + "="*80)
print("PART 2: DETERMINING OPTIMAL NUMBER OF CLUSTERS")
print("="*80)

# Test range of k values
k_range = range(2, 11)
inertias = []
silhouette_scores = []
davies_bouldin_scores = []

print("\nTesting k = 2 to 10 clusters...")
print("-" * 60)

for k in k_range:
    # Fit K-means
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10, max_iter=300)
    labels = kmeans.fit_predict(features_pca)

    # Calculate metrics
    inertia = kmeans.inertia_
    silhouette = silhouette_score(features_pca, labels)
    davies_bouldin = davies_bouldin_score(features_pca, labels)

    inertias.append(inertia)
    silhouette_scores.append(silhouette)
    davies_bouldin_scores.append(davies_bouldin)

    print(f"k={k:2d}: Inertia={inertia:8.2f}, Silhouette={silhouette:.4f}, Davies-Bouldin={davies_bouldin:.4f}")

print("-" * 60)
```

```python
# ============================================================================
# PART 3: ELBOW METHOD VISUALIZATION
# ============================================================================

print("\n" + "="*80)
print("PART 3: ELBOW METHOD ANALYSIS")
print("="*80)

# Calculate rate of decrease (elbow detection)
inertia_diff = np.diff(inertias)
inertia_diff_pct = (inertia_diff / inertias[:-1]) * 100

print("\nInertia decrease by adding each cluster:")
print("-" * 60)
for i, (k, decrease) in enumerate(zip(list(k_range)[1:], inertia_diff_pct)):
    print(f"k={k-1} - k={k}: {decrease:.2f}% decrease")
print("-" * 60)

# Find elbow using "knee" detection (largest second derivative)
if len(inertias) > 2:
    second_diff = np.diff(inertia_diff)
    elbow_k = np.argmax(second_diff) + 3  # +3 because of indexing
    print(f"\nElbow detected at k = {elbow_k}")
else:
    elbow_k = 3

# Visualize elbow plot
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Plot 1: Inertia (WCSS)
ax1 = axes[0]
ax1.plot(list(k_range), inertias, 'o-', linewidth=2, markersize=8, color='#2E86AB')
ax1.axvline(x=elbow_k, color='red', linestyle='--', linewidth=2,
            label=f'Elbow (k={elbow_k})')
ax1.set_xlabel('Number of Clusters (k)', fontsize=12, fontweight='bold')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', fontsize=12, fontweight='bold')
ax1.set_title('Elbow Method: Inertia', fontsize=14, fontweight='bold')
ax1.grid(True, alpha=0.3)
ax1.legend(fontsize=10)
ax1.set_xticks(list(k_range))

# Plot 2: Silhouette Score
ax2 = axes[1]
ax2.plot(list(k_range), silhouette_scores, 'o-', linewidth=2, markersize=8, color='#A23B72')
best_silhouette_k = list(k_range)[np.argmax(silhouette_scores)]
ax2.axvline(x=best_silhouette_k, color='red', linestyle='--', linewidth=2,
            label=f'Best (k={best_silhouette_k})')
ax2.set_xlabel('Number of Clusters (k)', fontsize=12, fontweight='bold')
ax2.set_ylabel('Silhouette Score', fontsize=12, fontweight='bold')
ax2.set_title('Silhouette Analysis', fontsize=14, fontweight='bold')
ax2.grid(True, alpha=0.3)
ax2.legend(fontsize=10)
ax2.set_xticks(list(k_range))
ax2.set_ylim([0, 1])

# Plot 3: Davies-Bouldin Index (lower is better)
ax3 = axes[2]
ax3.plot(list(k_range), davies_bouldin_scores, 'o-', linewidth=2, markersize=8, color='#F18F01')
best_db_k = list(k_range)[np.argmin(davies_bouldin_scores)]
ax3.axvline(x=best_db_k, color='red', linestyle='--', linewidth=2,
            label=f'Best (k={best_db_k})')
ax3.set_xlabel('Number of Clusters (k)', fontsize=12, fontweight='bold')
```

```python
# Cluster size distribution
cluster_counts = pd.Series(cluster_labels).value_counts().sort_index()
print(f"\nCluster Size Distribution:")
print("-" * 60)
for cluster_id, count in cluster_counts.items():
    percentage = (count / len(cluster_labels)) * 100
    print(f"  Cluster {cluster_id}: {count:3d} samples ({percentage:5.2f}%)")
print("-" * 60)

# ============================================================================
# PART 6: SILHOUETTE ANALYSIS PER CLUSTER
# ============================================================================

print("\n" + "="*80)
print("PART 6: DETAILED SILHOUETTE ANALYSIS")
print("="*80)

# Calculate silhouette scores for each sample
silhouette_vals = silhouette_samples(features_pca, cluster_labels)

# Silhouette plot
fig, ax = plt.subplots(figsize=(10, 8))

y_lower = 10
colors = plt.cm.Set2(np.linspace(0, 1, optimal_k))

for i in range(optimal_k):
    # Get silhouette scores for cluster i
    cluster_silhouette_vals = silhouette_vals[cluster_labels == i]
    cluster_silhouette_vals.sort()

    size_cluster_i = cluster_silhouette_vals.shape[0]
    y_upper = y_lower + size_cluster_i

    ax.fill_betweenx(np.arange(y_lower, y_upper),
                     0, cluster_silhouette_vals,
                     facecolor=colors[i], edgecolor=colors[i], alpha=0.7)

    # Label the silhouette plots with cluster numbers at the middle
    ax.text(-0.05, y_lower + 0.5 * size_cluster_i, f'Cluster {i}')

    y_lower = y_upper + 10

# Average silhouette score line
avg_silhouette = silhouette_score(features_pca, cluster_labels)
ax.axvline(x=avg_silhouette, color="red", linestyle="--", linewidth=2,
           label=f'Average: {avg_silhouette:.3f}')

ax.set_xlabel('Silhouette Coefficient', fontsize=12, fontweight='bold')
ax.set_ylabel('Cluster', fontsize=12, fontweight='bold')
ax.set_title(f'Silhouette Analysis (k={optimal_k})', fontsize=14, fontweight='bold')
ax.set_xlim([-0.2, 1])
ax.set_yticks([])
ax.legend(fontsize=10)
ax.grid(True, alpha=0.3, axis='x')

plt.tight_layout()
plt.savefig('kmeans_silhouette_analysis.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nSilhouette analysis plot saved: kmeans_silhouette_analysis.png")
```

```python
ax3.set_xlabel('Number of Clusters (k)', fontsize=12, fontweight='bold')
ax3.set_ylabel('Davies-Bouldin Index', fontsize=12, fontweight='bold')
ax3.set_title('Davies-Bouldin Index (Lower = Better)', fontsize=14, fontweight='bold')
ax3.grid(True, alpha=0.3)
ax3.legend(fontsize=10)
ax3.set_xticks(list(k_range))

plt.tight_layout()
plt.savefig('kmeans_optimal_k_selection.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nOptimal k selection plots saved: kmeans_optimal_k_selection.png")

# ============================================================================
# PART 4: FINAL CLUSTER SELECTION
# ============================================================================

print("\n" + "="*80)
print("PART 4: SELECTING OPTIMAL NUMBER OF CLUSTERS")
print("="*80)

print("\nCLUSTER SELECTION SUMMARY:")
print("-" * 60)
print(f"1. Elbow Method: k = {elbow_k}")
print(f"2. Silhouette Method: k = {best_silhouette_k} (score: {max(silhouette_scores):.4f})")
print(f"3. Davies-Bouldin Method: k = {best_db_k} (score: {min(davies_bouldin_scores):.4f})")
print(f"4. Domain Knowledge: k = 3-5 (WHO/Nepal water quality categories)")
print("-" * 60)

# Determine final k (prioritize silhouette + domain knowledge)
# Typically choose k that balances all metrics
optimal_k = best_silhouette_k

print(f"\n{'='*60}")
print(f"FINAL SELECTION: k = {optimal_k} clusters")
print(f"  - Rationale: Best silhouette score + interpretable categories")
print(f"  - Silhouette Score: {silhouette_scores[optimal_k-2]:.4f}")
print(f"  - Davies-Bouldin Index: {davies_bouldin_scores[optimal_k-2]:.4f}")
print(f"{'='*60}")

# ============================================================================
# PART 5: FIT FINAL K-MEANS MODEL
# ============================================================================

print("\n" + "="*80)
print("PART 5: FITTING FINAL K-MEANS MODEL")
print("="*80)

# Fit final model
kmeans_final = KMeans(n_clusters=optimal_k, random_state=42, n_init=10, max_iter=300)
cluster_labels = kmeans_final.fit_predict(features_pca)
centroids = kmeans_final.cluster_centers_

print(f"\n[] K-Means fitted with k={optimal_k} clusters")
print(f"  - Converged in {kmeans_final.n_iter_} iterations")
print(f"  - Final inertia: {kmeans_final.inertia_:.2f}")

# Add cluster labels to dataframes
full_df['cluster'] = cluster_labels
features_pca_df['cluster'] = cluster_labels
```

```python
# Print cluster-wise silhouette scores
print(f"\nCluster Quality (Silhouette Scores):")
print("-" * 60)
for i in range(optimal_k):
    cluster_silhouette_mean = silhouette_vals[cluster_labels == i].mean()
    print(f"  Cluster {i}: {cluster_silhouette_mean:.4f}")
print(f"  Overall Average: {avg_silhouette:.4f}")
print("-" * 60)

# ============================================================================
# PART 7: CLUSTER VISUALIZATION (2D PCA SPACE)
# ============================================================================

print("\n" + "="*80)
print("PART 7: CLUSTER VISUALIZATION IN PCA SPACE")
print("="*80)

fig, axes = plt.subplots(1, 2, figsize=(18, 8))

# Plot 1: Clusters colored
ax1 = axes[0]
scatter1 = ax1.scatter(features_pca[:, 0], features_pca[:, 1],
                       c=cluster_labels, cmap='Set2',
                       s=60, alpha=0.6, edgecolors='black', linewidth=0.5)

# Plot centroids
ax1.scatter(centroids[:, 0], centroids[:, 1],
            c='red', marker='X', s=300, edgecolors='black', linewidth=2,
            label='Centroids', zorder=10)

ax1.set_xlabel('PC1 (49.34% variance)', fontsize=12, fontweight='bold')
ax1.set_ylabel('PC2 (12.47% variance)', fontsize=12, fontweight='bold')
ax1.set_title(f'K-Means Clustering (k={optimal_k})', fontsize=14, fontweight='bold')
ax1.grid(True, alpha=0.3)
ax1.legend(fontsize=10)
ax1.axhline(y=0, color='k', linestyle='--', linewidth=0.5, alpha=0.3)
ax1.axvline(x=0, color='k', linestyle='--', linewidth=0.5, alpha=0.3)

# Add colorbar
cbar1 = plt.colorbar(scatter1, ax=ax1)
cbar1.set_label('Cluster', fontsize=10)

# Plot 2: Silhouette values colored
ax2 = axes[1]
scatter2 = ax2.scatter(features_pca[:, 0], features_pca[:, 1],
                       c=silhouette_vals, cmap='RdYlGn',
                       s=60, alpha=0.6, edgecolors='black', linewidth=0.5)

ax2.set_xlabel('PC1 (49.34% variance)', fontsize=12, fontweight='bold')
ax2.set_ylabel('PC2 (12.47% variance)', fontsize=12, fontweight='bold')
ax2.set_title('Cluster Quality (Silhouette Values)', fontsize=14, fontweight='bold')
ax2.grid(True, alpha=0.3)
ax2.axhline(y=0, color='k', linestyle='--', linewidth=0.5, alpha=0.3)
ax2.axvline(x=0, color='k', linestyle='--', linewidth=0.5, alpha=0.3)

# Add colorbar
cbar2 = plt.colorbar(scatter2, ax=ax2)
cbar2.set_label('Silhouette Score', fontsize=10)

plt.tight_layout()
plt.savefig('kmeans_clusters_2d.png', dpi=300, bbox_inches='tight')
```

```
plt.tight_layout()
plt.savefig('kmeans_clusters_2d.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nCluster visualization saved: kmeans_clusters_2d.png")

# ============================================================================
# PART 8: CLUSTER PROFILING (ORIGINAL FEATURES)
# ============================================================================

print("\n" + "="*80)
print("PART 8: CLUSTER PROFILING WITH ORIGINAL FEATURES")
print("="*80)

# Calculate mean feature values per cluster
cluster_profiles = full_df.groupby('cluster')[features_df.columns].mean()

print("\nCluster Profiles (Mean Feature Values):")
print("="*80)
print(cluster_profiles.round(2))
print("="*80)

# Calculate standard deviations
cluster_std = full_df.groupby('cluster')[features_df.columns].std()
print("\nCluster Variability (Standard Deviations):")
print("="*80)
print(cluster_std.round(2))
print("="*80)

# ============================================================================
# PART 9: RADAR CHART (CLUSTER COMPARISON)
# ============================================================================

print("\n" + "="*80)
print("PART 9: RADAR CHART VISUALIZATION")
print("="*80)

# Normalize cluster profiles for radar chart (0-1 scale)
from sklearn.preprocessing import MinMaxScaler
scaler_radar = MinMaxScaler()
cluster_profiles_normalized = pd.DataFrame(
    scaler_radar.fit_transform(cluster_profiles.T).T,
    columns=cluster_profiles.columns,
    index=cluster_profiles.index
)

# Create radar chart
num_vars = len(features_df.columns)
angles = np.linspace(0, 2 * np.pi, num_vars, endpoint=False).tolist()
angles += angles[:1]  # Complete the circle

fig, ax = plt.subplots(figsize=(12, 12), subplot_kw=dict(projection='polar'))

colors_radar = plt.cm.Set2(np.linspace(0, 1, optimal_k))

for i, (cluster_id, row) in enumerate(cluster_profiles_normalized.iterrows()):
    values = row.tolist()
    values += values[:1]  # Complete the circle

    ax.plot(angles, values, 'o-', linewidth=2, label=f'Cluster {cluster_id}',
            color=colors_radar[i])
    ax.fill(angles, values, alpha=0.15, color=colors_radar[i])
```

```
# ============================================================================
# PART 11: CLUSTER HEATMAP
# ============================================================================

print("\n" + "="*80)
print("PART 11: CLUSTER CHARACTERISTICS HEATMAP")
print("="*80)

fig, ax = plt.subplots(figsize=(12, 8))
sns.heatmap(cluster_profiles.T, annot=True, fmt='.2f', cmap='YlOrRd',
            cbar_kws={'label': 'Mean Value'}, linewidths=1, ax=ax)
ax.set_xlabel('Cluster', fontsize=12, fontweight='bold')
ax.set_ylabel('Water Quality Parameters', fontsize=12, fontweight='bold')
ax.set_title('Cluster Characteristics (Mean Feature Values)',
             fontsize=14, fontweight='bold', pad=20)
plt.tight_layout()
plt.savefig('kmeans_cluster_heatmap.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nCluster heatmap saved: kmeans_cluster_heatmap.png")

# ============================================================================
# PART 12: SAVE OUTPUTS
# ============================================================================

print("\n" + "="*80)
print("PART 12: SAVING OUTPUTS")
print("="*80)

# Save clustered data
full_df.to_excel('/content/drive/MyDrive/Mahakali_River_Data/data_with_clusters.xlsx',
                 index=False)
print("Saved: data_with_clusters.xlsx (full dataset with cluster labels)")

# Save cluster profiles
cluster_profiles.to_csv('/content/cluster_profiles.csv')
print("Saved: cluster_profiles.csv (mean feature values per cluster)")

# Save K-means model
import joblib
joblib.dump(kmeans_final, '/content/kmeans_model.pkl')
print("Saved: kmeans_model.pkl (trained K-means model)")

# Save cluster assignments
cluster_assignments = pd.DataFrame({
    'sample_id': range(len(cluster_labels)),
    'cluster': cluster_labels,
    'silhouette_score': silhouette_vals
})
cluster_assignments.to_csv('/content/cluster_assignments.csv', index=False)
print("Saved: cluster_assignments.csv (cluster labels + quality scores)")

print("\n" + "="*80)
print("K-MEANS CLUSTERING COMPLETE!")
print("="*80)

print(f"\nFINAL SUMMARY:")
print(f"  - Number of clusters: {optimal_k}")
print(f"  - Silhouette score: {avg_silhouette:.4f}")
print(f"  - Davies-Bouldin index: {davies_bouldin_scores[optimal_k-2]:.4f}")
print(f"  - All 720 samples assigned to clusters")
```

```
for i, (cluster_id, row) in enumerate(cluster_profiles_normalized.iterrows()):
    values = row.tolist()
    values += values[:1]  # Complete the circle

    ax.plot(angles, values, 'o-', linewidth=2, label=f'Cluster {cluster_id}',
            color=colors_radar[i])
    ax.fill(angles, values, alpha=0.15, color=colors_radar[i])

# Set labels
ax.set_xticks(angles[:-1])
ax.set_xticklabels(features_df.columns, size=10)
ax.set_ylim(0, 1)
ax.set_yticks([0.25, 0.5, 0.75, 1.0])
ax.set_yticklabels(['0.25', '0.5', '0.75', '1.0'], size=8)
ax.set_title(f'Cluster Profiles (Normalized Features)',
             fontsize=14, fontweight='bold', pad=20)
ax.legend(loc='upper right', bbox_to_anchor=(1.3, 1.1), fontsize=10)
ax.grid(True)

plt.tight_layout()
plt.savefig('kmeans_cluster_radar.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nRadar chart saved: kmeans_cluster_radar.png")

# ============================================================================
# PART 10: SPATIAL-TEMPORAL ANALYSIS
# ============================================================================

print("\n" + "="*80)
print("PART 10: SPATIAL-TEMPORAL DISTRIBUTION")
print("="*80)

# District distribution
print("\nCluster Distribution by District:")
print("-" * 60)
district_cluster = pd.crosstab(full_df['district'], full_df['cluster'],
                               normalize='index') * 100
print(district_cluster.round(2))
print("-" * 60)

# Station distribution
print("\nCluster Distribution by Station:")
print("-" * 60)
station_cluster = pd.crosstab(full_df['station'], full_df['cluster'],
                              normalize='index') * 100
print(station_cluster.round(2))
print("-" * 60)

# Temporal distribution (if date available)
if 'date' in full_df.columns and full_df['date'].notna().any():
    full_df['month'] = pd.to_datetime(full_df['date']).dt.month
    print("\nCluster Distribution by Month:")
    print("-" * 60)
    month_cluster = pd.crosstab(full_df['month'], full_df['cluster'],
                                normalize='index') * 100
    print(month_cluster.round(2))
    print("-" * 60)

# ============================================================================
# PART 11: CLUSTER HEATMAP
```

```
# ============================================================================
# PART 3: CALCULATE WATER QUALITY INDEX (WQI) - CORRECTED
# ============================================================================

print("\n" + "="*80)
print("PART 3: WATER QUALITY INDEX (WQI) CALCULATION")
print("="*80)

# WHO Drinking Water Guidelines (2017) + Nepal Standards (NS 2060:2062)
standards = {
    'parameter': ['temperature', 'ph', 'total_hardness', 'chloride',
                  'ammonia', 'phosphate', 'nitrate', 'iron',
                  'free_residual_chlorin', 'coliform'],

    # Ideal values (pristine/excellent quality)
    'ideal': [20, 7.0, 100, 30, 0.0, 0.0, 0.0, 0.0, 0.2, 0.0],

    # Permissible limits (WHO/Nepal standards)
    'permissible': [None, 8.5, 500, 250, 1.5, None, 50, 0.3, 0.5, 0],

    # Lower limits (for pH, temperature)
    'lower_limit': [None, 6.5, None, None, None, None, None, None, None, None],

    # Units
    'unit': ['°C', 'pH', 'mg/L', 'mg/L', 'mg/L', 'mg/L', 'mg/L', 'mg/L', 'mg/L', 'MPN/100mL'],

    # Weights for WQI calculation (based on health significance)
    'weight': [0.05, 0.15, 0.08, 0.08, 0.12, 0.10, 0.12, 0.10, 0.05, 0.15]
}

standards_df = pd.DataFrame(standards)

print("\nWHO/Nepal Water Quality Standards:")
print("="*80)
print(standards_df.to_string(index=False))
print("="*80)

def calculate_quality_rating(value, ideal, permissible, lower_limit=None, param_name=None):
    """
    Calculate quality rating (Qi) for a parameter.

    Formula: Qi = [(Vi - Vi_ideal) / (Si - Vi_ideal)] x 100

    Special handling for parameters where standard = 0 (like coliform)
    """
    # Handle NaN values
    if pd.isna(value):
        return 100  # Penalize missing data

    # Special case: Coliform (permissible = 0, any detection is violation)
    if param_name == 'coliform':
        if value <= 0.5:  # Encoding for "no detection"
            return 0  # Excellent
        elif value <= 100:  # White (moderate)
            return 50  # Moderate quality
        else:  # Black (high contamination)
            return 100  # Poor quality

    # Special case: Parameters without permissible limits
    if pd.isna(permissible):
        # For temperature and phosphate (no clear WHO limit)
        if ideal == 0:
```

```
def calculate_quality_rating(value, ideal, permissible, lower_limit=None, param_name=None):
    if pd.isna(permissible):
        if ideal == 0:
            return min((value / 1.0) * 100, 100)  # Cap at 100
        else:
            # Calculate deviation from ideal
            deviation_pct = abs(value - ideal) / ideal * 100
            return min(deviation_pct, 100)  # Cap at 100

    # Special case: pH (has both lower and upper limits)
    if param_name == 'ph':
        if value < lower_limit:
            # Below acceptable range
            qi = abs(value - ideal) / abs(lower_limit - ideal) * 100
        elif value > permissible:
            # Above acceptable range
            qi = abs(value - ideal) / abs(permissible - ideal) * 100
        else:
            # Within acceptable range - calculate proportional quality
            if value < ideal:
                qi = abs(value - ideal) / abs(lower_limit - ideal) * 50
            else:
                qi = abs(value - ideal) / abs(permissible - ideal) * 50
        return qi

    # Standard formula for other parameters
    # Qi = [(Vi - Vi_ideal) / (Si - Vi_ideal)] × 100

    if permissible == ideal:
        return 0 if value == ideal else 100

    # Calculate quality rating
    qi = abs(value - ideal) / abs(permissible - ideal) * 100

    # Cap extreme values
    return min(qi, 150)  # Allow some values above 100 for severe pollution


def calculate_wqi(row, standards_df):
    """
    Calculate Water Quality Index for a sample.

    WQI = Σ(Wi × Qi)
    """
    wqi = 0

    for idx, param_row in standards_df.iterrows():
        param_name = param_row['parameter']

        if param_name in row.index:
            value = row[param_name]
            ideal = param_row['ideal']
            permissible = param_row['permissible']
            lower_limit = param_row['lower_limit']
            weight = param_row['weight']

            # Calculate quality rating
            qi = calculate_quality_rating(value, ideal, permissible, lower_limit, param_name)

            # Add weighted contribution to WQI
            wqi += weight * qi
```

```
def classify_wqi(wqi_score):
    if wqi_score <= 25:
        return 'Excellent'
    elif wqi_score <= 50:
        return 'Good'
    elif wqi_score <= 75:
        return 'Poor'
    elif wqi_score <= 100:
        return 'Very Poor'
    else:
        return 'Unsuitable'

# Apply classification
full_df['wqi_category'] = full_df['wqi'].apply(classify_wqi)

print("\nWQI Classification Scale:")
print("-"*60)
print("  Excellent:    0-25   (High quality, minimal treatment needed)")
print("  Good:        26-50   (Acceptable, basic treatment needed)")
print("  Poor:        51-75   (Contaminated, extensive treatment needed)")
print("  Very Poor:   76-100  (Highly polluted, not recommended for use)")
print("  Unsuitable:  >100    (Severe pollution, unsuitable for any use)")
print("-"*60)

# Distribution of WQI categories
print("\nOverall WQI Category Distribution:")
print("="*60)
wqi_category_dist = full_df['wqi_category'].value_counts().sort_index()
for category, count in wqi_category_dist.items():
    pct = (count / len(full_df)) * 100
    print(f"  {category:12s}: {count:3d} samples ({pct:5.2f}%)")
print("="*60)

# WQI categories by cluster
print("\nWQI Categories by Cluster:")
print("="*60)
cluster_wqi_crosstab = pd.crosstab(full_df['cluster'], full_df['wqi_category'], normalize='index') * 100
print(cluster_wqi_crosstab.round(2))
print("="*60)

# ============================================================================
# PART 5: ASSIGN FINAL CLUSTER LABELS
# ============================================================================

print("\n" + "="*80)
print("PART 5: FINAL CLUSTER LABEL ASSIGNMENT")
print("="*80)

# Calculate mean WQI per cluster
cluster_mean_wqi = full_df.groupby('cluster')['wqi'].mean()

# Assign labels based on mean WQI
cluster_labels = {}
for cluster_id, mean_wqi in cluster_mean_wqi.items():
    category = classify_wqi(mean_wqi)
    cluster_labels[cluster_id] = category
    print(f"Cluster {cluster_id}: Mean WQI = {mean_wqi:.2f} → Label: '{category}'")

# Add cluster label to dataframe
full_df['cluster_label'] = full_df['cluster'].map(cluster_labels)
```

```
# Calculate WQI for each sample
print("\nCalculating WQI for all samples...")
features_cols = ['temperature', 'ph', 'total_hardness', 'chloride',
                 'ammonia', 'phosphate', 'nitrate', 'iron',
                 'free_residual_chlorin', 'coliform']

full_df['wqi'] = full_df[features_cols].apply(
    lambda row: calculate_wqi(row, standards_df), axis=1
)

print(f"✓ WQI calculated for {len(full_df)} samples")

# Show sample calculations for verification
print("\n" + "="*60)
print("SAMPLE WQI CALCULATIONS (first 5 samples):")
print("="*60)
sample_display = full_df[['district', 'station', 'cluster'] + features_cols + ['wqi']].head()
print(sample_display.to_string(index=False))
print("="*60)

# Calculate WQI statistics
print("\nOverall WQI Statistics:")
print("-" * 60)
print(f"  Mean: {full_df['wqi'].mean():.2f}")
print(f"  Median: {full_df['wqi'].median():.2f}")
print(f"  Std Dev: {full_df['wqi'].std():.2f}")
print(f"  Min: {full_df['wqi'].min():.2f}")
print(f"  Max: {full_df['wqi'].max():.2f}")
print("-" * 60)

# Calculate WQI statistics by cluster
print("\nWQI Statistics by Cluster:")
print("="*60)
wqi_by_cluster = full_df.groupby('cluster')['wqi'].agg(['count', 'mean', 'std', 'min', 'max'])
print(wqi_by_cluster.round(2))
print("="*60)

# Verify no NaN values
nan_count = full_df['wqi'].isna().sum()
if nan_count > 0:
    print(f"\nWarning: {nan_count} samples have NaN WQI values")
    print("Investigating samples with NaN:")
    nan_samples = full_df[full_df['wqi'].isna()][features_cols]
    print(nan_samples.head())
else:
    print(f"\nAll {len(full_df)} samples have valid WQI values")

# ============================================================================
# PART 4: ASSIGN WQI-BASED QUALITY CATEGORIES
# ============================================================================

print("\n" + "="*80)
print("PART 4: WQI-BASED CLASSIFICATION")
print("="*80)

# WQI classification scheme (standard international scale)
def classify_wqi(wqi_score):
    """
    Classify water quality based on WQI score.
    """
    if wqi_score <= 25:
```

```
print("="*60)
for cluster_id, label in cluster_labels.items():
    count = (full_df['cluster'] == cluster_id).sum()
    percentage = (count / len(full_df)) * 100
    mean_wqi = cluster_mean_wqi[cluster_id]
    print(f"  Cluster {cluster_id} → '{label}' (WQI: {mean_wqi:.2f}, {count} samples, {percentage:.2f}%)")
print("="*60)

# ============================================================================
# PART 6: VISUALIZATIONS
# ============================================================================

print("\n" + "="*80)
print("PART 6: VISUALIZATION")
print("="*80)

# Create 4-panel visualization
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Subplot 1: WQI histogram by cluster
ax1 = axes[0, 0]
colors_cluster = ['#e74c3c', '#3498db']  # Red for Cluster 0, Blue for Cluster 1

for idx, cluster_id in enumerate(sorted(full_df['cluster'].unique())):
    cluster_wqi = full_df[full_df['cluster'] == cluster_id]['wqi']
    label_text = f"Cluster {cluster_id} ({cluster_labels[cluster_id]})"
    ax1.hist(cluster_wqi, bins=30, alpha=0.6, label=label_text,
             edgecolor='black', color=colors_cluster[idx])

ax1.axvline(25, color='green', linestyle='--', linewidth=2, alpha=0.7)
ax1.axvline(50, color='orange', linestyle='--', linewidth=2, alpha=0.7)
ax1.axvline(75, color='red', linestyle='--', linewidth=2, alpha=0.7)

ax1.text(12.5, ax1.get_ylim()[1]*0.9, 'Excellent', ha='center', fontsize=9, color='green')
ax1.text(37.5, ax1.get_ylim()[1]*0.9, 'Good', ha='center', fontsize=9, color='orange')
ax1.text(62.5, ax1.get_ylim()[1]*0.9, 'Poor', ha='center', fontsize=9, color='red')

ax1.set_xlabel('Water Quality Index (WQI)', fontsize=12, fontweight='bold')
ax1.set_ylabel('Frequency', fontsize=12, fontweight='bold')
ax1.set_title('WQI Distribution by Cluster', fontsize=14, fontweight='bold')
ax1.legend(fontsize=10)
ax1.grid(True, alpha=0.3)

# Subplot 2: WQI boxplot by cluster
ax2 = axes[0, 1]
cluster_data = [full_df[full_df['cluster'] == cid]['wqi'].values
                for cid in sorted(full_df['cluster'].unique())]
bp = ax2.boxplot(cluster_data,
                 labels=[f"Cluster {cid}\n({cluster_labels[cid]})"
                         for cid in sorted(full_df['cluster'].unique())],
                 patch_artist=True)

for patch, color in zip(bp['boxes'], colors_cluster):
    patch.set_facecolor(color)
    patch.set_alpha(0.6)

ax2.axhline(25, color='green', linestyle='--', linewidth=1, alpha=0.5, label='Excellent/Good')
ax2.axhline(50, color='orange', linestyle='--', linewidth=1, alpha=0.5, label='Good/Poor')
ax2.axhline(75, color='red', linestyle='--', linewidth=1, alpha=0.5, label='Poor/Very Poor')
ax2.set_ylabel('Water Quality Index (WQI)', fontsize=12, fontweight='bold')
ax2.set_title('WQI Comparison Across Clusters', fontsize=14, fontweight='bold')
ax2.legend(fontsize=8, loc='upper right')
```

```
ax2.legend(fontsize=8, loc='upper right')
ax2.grid(True, alpha=0.3, axis='y')

# Subplot 3: Category distribution pie chart
ax3 = axes[1, 0]
category_counts = full_df['wqi_category'].value_counts()
colors_pie = {'Excellent': '#2ecc71', 'Good': '#3498db', 'Poor': '#e67e22',
              'Very Poor': '#e74c3c', 'Unsuitable': '#95a5a6'}
pie_colors = [colors_pie.get(cat, '#95a5a6') for cat in category_counts.index]

wedges, texts, autotexts = ax3.pie(category_counts.values, labels=category_counts.index,
                                   autopct='%1.1f%%', colors=pie_colors, startangle=90)
for autotext in autotexts:
    autotext.set_color('white')
    autotext.set_fontweight('bold')
    autotext.set_fontsize(11)

ax3.set_title('Overall Water Quality Distribution\n(All 720 Samples)',
              fontsize=14, fontweight='bold')

# Subplot 4: Cluster-wise category stacked bar
ax4 = axes[1, 1]
cluster_category = pd.crosstab(full_df['cluster'], full_df['wqi_category'], normalize='index') * 100

# Reorder columns to match quality scale
category_order = ['Excellent', 'Good', 'Poor', 'Very Poor', 'Unsuitable']
existing_categories = [cat for cat in category_order if cat in cluster_category.columns]
cluster_category = cluster_category[existing_categories]

cluster_category.plot(kind='bar', stacked=True, ax=ax4,
                      color=[colors_pie.get(cat, '#95a5a6') for cat in existing_categories],
                      edgecolor='black', linewidth=0.5)
ax4.set_xlabel('Cluster', fontsize=12, fontweight='bold')
ax4.set_ylabel('Percentage (%)', fontsize=12, fontweight='bold')
ax4.set_title('Water Quality Categories by Cluster', fontsize=14, fontweight='bold')
ax4.set_xticklabels([f'Cluster {cid}\n({cluster_labels[cid]})'
                     for cid in cluster_category.index], rotation=0)
ax4.legend(title='WQI Category', fontsize=9, title_fontsize=10, loc='upper right')
ax4.grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.savefig('wqi_analysis.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nWQI analysis visualization saved: wqi_analysis.png")

# ============================================================================
# PART 7: SAVE LABELED DATASET
# ============================================================================

print("\n" + "="*80)
print("PART 7: SAVING LABELED DATASET")
print("="*80)

# Save complete dataset with all labels
output_path = '/content/drive/MyDrive/Mahakali_River_Data/data_with_labels.xlsx'
full_df.to_excel(output_path, index=False)
print(f"Saved labeled dataset: {output_path}")

# Save label mapping
label_mapping = pd.DataFrame({
```

```
# Check for class imbalance
imbalance_ratio = class_dist.max() / class_dist.min()
print(f"\nClass imbalance ratio: {imbalance_ratio:.2f}:1")
if imbalance_ratio > 1.5:
    print("Moderate imbalance detected - will use SMOTE")
    use_smote = True
else:
    print("Classes are relatively balanced")
    use_smote = False

# ============================================================================
# PART 2: TRAIN-TEST SPLIT
# ============================================================================

print("\n" + "="*80)
print("PART 2: TRAIN-TEST SPLIT")
print("="*80)

# Encode target labels (Poor=0, Good=1)
le = LabelEncoder()
y_encoded = le.fit_transform(y)

print(f"\nLabel encoding:")
for i, label in enumerate(le.classes_):
    print(f"  '{label}' → {i}")

# Split data (80% train, 20% test) with stratification
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded,
    test_size=0.2,
    random_state=42,
    stratify=y_encoded
)

print(f"\nTrain-test split complete:")
print(f"  - Training set: {X_train.shape[0]} samples ({X_train.shape[0]/len(X)*100:.1f}%)")
print(f"  - Test set: {X_test.shape[0]} samples ({X_test.shape[0]/len(X)*100:.1f}%)")

# Check class distribution in splits
print("\nClass distribution in splits:")
print("-" * 60)
train_dist = pd.Series(y_train).value_counts()
test_dist = pd.Series(y_test).value_counts()

for class_id in sorted(train_dist.index):
    label = le.inverse_transform([class_id])[0]
    train_count = train_dist.get(class_id, 0)
    test_count = test_dist.get(class_id, 0)
    print(f"  {label:10s}: Train={train_count:3d} ({train_count/len(y_train)*100:5.1f}%), " +
          f"Test={test_count:3d} ({test_count/len(y_test)*100:5.1f}%)")
print("-" * 60)

# ============================================================================
# PART 3: FEATURE SCALING
# ============================================================================

print("\n" + "="*80)
print("PART 3: FEATURE SCALING")
print("="*80)

# Scale features (important for SVM and k-NN)
```

```
# Save label mapping
label_mapping = pd.DataFrame({
    'cluster': list(cluster_labels.keys()),
    'label': list(cluster_labels.values()),
    'mean_wqi': [cluster_mean_wqi[cid] for cid in cluster_labels.keys()],
    'sample_count': [sum(full_df['cluster'] == cid) for cid in cluster_labels.keys()]
})
label_mapping.to_csv('/content/cluster_label_mapping.csv', index=False)
print("✓ Saved label mapping: cluster_label_mapping.csv")

# Save WQI summary statistics
wqi_summary = full_df.groupby(['cluster', 'cluster_label'])['wqi'].describe()
wqi_summary.to_csv('/content/wqi_summary_statistics.csv')
print("✓ Saved WQI statistics: wqi_summary_statistics.csv")

print("\n" + "="*80)
print("LABEL GENERATION COMPLETE!")
print("="*80)

print(f"\nFINAL SUMMARY:")
print(f"  - Total samples: {len(full_df)}")
print(f"  - Clusters labeled: {len(cluster_labels)}")
print(f"  - Quality categories identified: {full_df['wqi_category'].nunique()}")
print(f"  - Mean WQI (overall): {full_df['wqi'].mean():.2f}")
print(f"  - WQI range: {full_df['wqi'].min():.2f} - {full_df['wqi'].max():.2f}")

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

"""
PHASE 4: SUPERVISED CLASSIFICATION
Random Forest, SVM, and k-NN for Water Quality Prediction
"""

# Set plot style
plt.style.use('seaborn-v0_8-darkgrid')
sns.set_palette("Set2")

print("="*80)
print("PHASE 4: SUPERVISED CLASSIFICATION")
print("="*80)

# ============================================================================
# PART 1: LOAD LABELED DATA
# ============================================================================

print("\n" + "="*80)
print("PART 1: LOADING LABELED DATA")
print("="*80)

# Load the labeled dataset from Phase 3
full_df = pd.read_excel('/content/drive/MyDrive/Mahakali_River_Data/data_with_labels.xlsx')

# Define features and target
feature_columns = ['temperature', 'ph', 'total_hardness', 'chloride',
                   'ammonia', 'phosphate', 'nitrate', 'iron',
                   'free_residual_chlorin', 'coliform']

X = full_df[feature_columns].copy()
y = full_df['cluster_label'].copy()

print(f"\nLoaded dataset: {full_df.shape}")
```

```
# Scale features (important for SVM and k-NN)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("\nFeatures scaled using StandardScaler")
print(f"  - Training set scaled: {X_train_scaled.shape}")
print(f"  - Test set scaled: {X_test_scaled.shape}")

# Save scaler
joblib.dump(scaler, '/content/classifier_scaler.pkl')
print("Scaler saved: classifier_scaler.pkl")

# ============================================================================
# PART 4: HANDLE CLASS IMBALANCE WITH SMOTE (IF NEEDED)
# ============================================================================

if use_smote:
    print("\n" + "="*80)
    print("PART 4: HANDLING CLASS IMBALANCE WITH SMOTE")
    print("="*80)

    smote = SMOTE(random_state=42)
    X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scaled, y_train)

    print(f"\nSMOTE applied:")
    print(f"  - Before: {X_train_scaled.shape[0]} samples")
    print(f"  - After: {X_train_resampled.shape[0]} samples")

    print("\nClass distribution after SMOTE:")
    resampled_dist = pd.Series(y_train_resampled).value_counts()
    for class_id in sorted(resampled_dist.index):
        label = le.inverse_transform([class_id])[0]
        count = resampled_dist[class_id]
        print(f"  {label:10s}: {count:3d} samples")

    # Use resampled data for training
    X_train_final = X_train_resampled
    y_train_final = y_train_resampled
else:
    print("\nSMOTE not needed - using original training data")
    X_train_final = X_train_scaled
    y_train_final = y_train

# ============================================================================
# PART 5: MODEL 1 - RANDOM FOREST
# ============================================================================

print("\n" + "="*80)
print("PART 5: RANDOM FOREST CLASSIFIER")
print("="*80)

print("\nTraining Random Forest...")

# Define Random Forest with optimized parameters
rf_model = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    min_samples_split=5,
    min_samples_leaf=2,
    random_state=42,
```

```
rf_model = RandomForestClassifier(
    ...)
)

# Train model
rf_model.fit(X_train_final, y_train_final)
print("Random Forest trained")

# Predictions
y_train_pred_rf = rf_model.predict(X_train_scaled)
y_test_pred_rf = rf_model.predict(X_test_scaled)
y_test_proba_rf = rf_model.predict_proba(X_test_scaled)[:, 1]

# Calculate metrics
rf_train_acc = accuracy_score(y_train, y_train_pred_rf)
rf_test_acc = accuracy_score(y_test, y_test_pred_rf)
rf_precision = precision_score(y_test, y_test_pred_rf, average='weighted')
rf_recall = recall_score(y_test, y_test_pred_rf, average='weighted')
rf_f1 = f1_score(y_test, y_test_pred_rf, average='weighted')

print(f"\nRandom Forest Performance:")
print("-" * 60)
print(f"  Training Accuracy: {rf_train_acc*100:.2f}%")
print(f"  Test Accuracy:     {rf_test_acc*100:.2f}%")
print(f"  Precision:         {rf_precision*100:.2f}%")
print(f"  Recall:            {rf_recall*100:.2f}%")
print(f"  F1-Score:          {rf_f1*100:.2f}%")
print("-" * 60)

# Feature importance
feature_importance = pd.DataFrame({
    'feature': feature_columns,
    'importance': rf_model.feature_importances_
}).sort_values('importance', ascending=False)

print("\nFeature Importance (Top 5):")
print("-" * 60)
for idx, row in feature_importance.head(5).iterrows():
    print(f"  {row['feature']:25s}: {row['importance']:.4f}")
print("-" * 60)

# Save model
joblib.dump(rf_model, '/content/random_forest_model.pkl')
print("\nRandom Forest model saved: random_forest_model.pkl")

# ============================================================================
# PART 6: MODEL 2 - SUPPORT VECTOR MACHINE (SVM)
# ============================================================================

print("\n" + "="*80)
print("PART 6: SUPPORT VECTOR MACHINE (SVM)")
print("="*80)

print("\nTraining SVM with RBF kernel...")

# Define SVM with RBF kernel
svm_model = SVC(
    kernel='rbf',
    C=10,
    gamma='scale',
    probability=True,  # Enable probability estimates for ROC curve
    random_state=42
)

# Train model
svm_model.fit(X_train_final, y_train_final)
print("SVM trained")

# Predictions
y_train_pred_svm = svm_model.predict(X_train_scaled)
y_test_pred_svm = svm_model.predict(X_test_scaled)
y_test_proba_svm = svm_model.predict_proba(X_test_scaled)[:, 1]

# Calculate metrics
svm_train_acc = accuracy_score(y_train, y_train_pred_svm)
svm_test_acc = accuracy_score(y_test, y_test_pred_svm)
svm_precision = precision_score(y_test, y_test_pred_svm, average='weighted')
svm_recall = recall_score(y_test, y_test_pred_svm, average='weighted')
svm_f1 = f1_score(y_test, y_test_pred_svm, average='weighted')

print(f"\nSVM Performance:")
print("-" * 60)
print(f"  Training Accuracy: {svm_train_acc*100:.2f}%")
print(f"  Test Accuracy:     {svm_test_acc*100:.2f}%")
print(f"  Precision:         {svm_precision*100:.2f}%")
print(f"  Recall:            {svm_recall*100:.2f}%")
print(f"  F1-Score:          {svm_f1*100:.2f}%")
print("-" * 60)

# Save model
joblib.dump(svm_model, '/content/svm_model.pkl')
print("\n SVM model saved: svm_model.pkl")

# ============================================================================
# PART 7: MODEL 3 - K-NEAREST NEIGHBORS (k-NN)
# ============================================================================

print("\n" + "="*80)
print("PART 7: K-NEAREST NEIGHBORS (k-NN)")
print("="*80)

print("\nFinding optimal k value...")

# Test different k values
k_range = range(3, 16, 2)
k_scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_final, y_train_final)
    score = knn.score(X_test_scaled, y_test)
    k_scores.append(score)
    print(f"  k={k:2d}: Test Accuracy = {score*100:.2f}%")

# Select best k
best_k = list(k_range)[np.argmax(k_scores)]
best_k_score = max(k_scores)

print(f"\nOptimal k = {best_k} (Accuracy: {best_k_score*100:.2f}%)")

# Train final k-NN model with best k
knn_model = KNeighborsClassifier(n_neighbors=best_k)
knn_model.fit(X_train_final, y_train_final)
print(f"k-NN trained with k={best_k}")

# Predictions
y_train_pred_knn = knn_model.predict(X_train_scaled)
y_test_pred_knn = knn_model.predict(X_test_scaled)
y_test_proba_knn = knn_model.predict_proba(X_test_scaled)[:, 1]

# Calculate metrics
knn_train_acc = accuracy_score(y_train, y_train_pred_knn)
knn_test_acc = accuracy_score(y_test, y_test_pred_knn)
knn_precision = precision_score(y_test, y_test_pred_knn, average='weighted')
knn_recall = recall_score(y_test, y_test_pred_knn, average='weighted')
knn_f1 = f1_score(y_test, y_test_pred_knn, average='weighted')

print(f"\nk-NN Performance:")
print("-" * 60)
print(f"  Training Accuracy: {knn_train_acc*100:.2f}%")
print(f"  Test Accuracy:     {knn_test_acc*100:.2f}%")
print(f"  Precision:         {knn_precision*100:.2f}%")
print(f"  Recall:            {knn_recall*100:.2f}%")
print(f"  F1-Score:          {knn_f1*100:.2f}%")
print("-" * 60)

# Save model
joblib.dump(knn_model, '/content/knn_model.pkl')
print(f"\nk-NN model saved: knn_model.pkl")

# ============================================================================
# PART 8: 5-FOLD CROSS-VALIDATION
# ============================================================================

print("\n" + "="*80)
print("PART 8: 5-FOLD CROSS-VALIDATION")
print("="*80)

# Define stratified k-fold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

print("\nPerforming 5-fold cross-validation...")
print("-" * 60)

# Cross-validation for Random Forest
rf_cv_scores = cross_val_score(rf_model, X_train_scaled, y_train, cv=skf, scoring='accuracy')
print(f"Random Forest CV: {rf_cv_scores.mean()*100:.2f}% (+/- {rf_cv_scores.std()*100:.2f}%)")

# Cross-validation for SVM
svm_cv_scores = cross_val_score(svm_model, X_train_scaled, y_train, cv=skf, scoring='accuracy')
print(f"SVM CV:           {svm_cv_scores.mean()*100:.2f}% (+/- {svm_cv_scores.std()*100:.2f}%)")

# Cross-validation for k-NN
knn_cv_scores = cross_val_score(knn_model, X_train_scaled, y_train, cv=skf, scoring='accuracy')
print(f"k-NN CV:          {knn_cv_scores.mean()*100:.2f}% (+/- {knn_cv_scores.std()*100:.2f}%)")
print("-" * 60)

# ============================================================================
# PART 9: MODEL COMPARISON
# ============================================================================

print("\n" + "="*80)
print("PART 9: MODEL COMPARISON")
print("="*80)

# Create comparison table
comparison_df = pd.DataFrame({
    'Model': ['Random Forest', 'SVM (RBF)', 'k-NN'],
    'Test Accuracy (%)': [rf_test_acc*100, svm_test_acc*100, knn_test_acc*100],
    'Precision (%)': [rf_precision*100, svm_precision*100, knn_precision*100],
    'Recall (%)': [rf_recall*100, svm_recall*100, knn_recall*100],
    'F1-Score (%)': [rf_f1*100, svm_f1*100, knn_f1*100],
    'CV Mean (%)': [rf_cv_scores.mean()*100, svm_cv_scores.mean()*100, knn_cv_scores.mean()*100],
    'CV Std (%)': [rf_cv_scores.std()*100, svm_cv_scores.std()*100, knn_cv_scores.std()*100]
})

print("\nMODEL PERFORMANCE COMPARISON:")
print("="*80)
print(comparison_df.to_string(index=False))
print("="*80)

# Identify best model
best_model_idx = comparison_df['Test Accuracy (%)'].idxmax()
best_model_name = comparison_df.loc[best_model_idx, 'Model']
best_model_acc = comparison_df.loc[best_model_idx, 'Test Accuracy (%)']

print(f"\nHIGH ACCURACY MODEL: {best_model_name} (Test Accuracy: {best_model_acc:.2f}%)")

# Save comparison table
comparison_df.to_csv('/content/model_comparison.csv', index=False)
print("Model comparison saved: model_comparison.csv")

# ============================================================================
# PART 10: CONFUSION MATRICES
# ============================================================================

print("\n" + "="*80)
print("PART 10: CONFUSION MATRICES")
print("="*80)

# Calculate confusion matrices
cm_rf = confusion_matrix(y_test, y_test_pred_rf)
cm_svm = confusion_matrix(y_test, y_test_pred_svm)
cm_knn = confusion_matrix(y_test, y_test_pred_knn)

# Plot confusion matrices
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

models_cm = [
    ('Random Forest', cm_rf, rf_test_acc),
    ('SVM (RBF)', cm_svm, svm_test_acc),
    ('k-NN', cm_knn, knn_test_acc)
]

for ax, (model_name, cm, acc) in zip(axes, models_cm):
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax,
                xticklabels=le.classes_, yticklabels=le.classes_,
                cbar_kws={'label': 'Count'})
```

```
for ax, (model_name, cm, acc) in zip(axes, models_cm):
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax,
                xticklabels=le.classes_, yticklabels=le.classes_,
                cbar_kws={'label': 'Count'})
    ax.set_xlabel('Predicted Label', fontsize=12, fontweight='bold')
    ax.set_ylabel('True Label', fontsize=12, fontweight='bold')
    ax.set_title(f'{model_name}\n(Accuracy: {acc*100:.2f}%)',
                 fontsize=14, fontweight='bold')

plt.tight_layout()
plt.savefig('confusion_matrices.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nConfusion matrices saved: confusion_matrices.png")

# Print classification reports
print("\n" + "="*60)
print("DETAILED CLASSIFICATION REPORTS")
print("="*60)

for model_name, y_pred in [('Random Forest', y_test_pred_rf),
                           ('SVM', y_test_pred_svm),
                           ('k-NN', y_test_pred_knn)]:
    print(f"\n{model_name}:")
    print("-" * 60)
    print(classification_report(y_test, y_pred, target_names=le.classes_))

# ============================================================================
# PART 11: ROC CURVES
# ============================================================================

print("\n" + "="*80)
print("PART 11: ROC CURVES")
print("="*80)

# Calculate ROC curves
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_test_proba_rf)
fpr_svm, tpr_svm, _ = roc_curve(y_test, y_test_proba_svm)
fpr_knn, tpr_knn, _ = roc_curve(y_test, y_test_proba_knn)

# Calculate AUC scores
auc_rf = auc(fpr_rf, tpr_rf)
auc_svm = auc(fpr_svm, tpr_svm)
auc_knn = auc(fpr_knn, tpr_knn)

print(f"\nAUC-ROC Scores:")
print("-" * 60)
print(f"  Random Forest: {auc_rf:.4f}")
print(f"  SVM:           {auc_svm:.4f}")
print(f"  k-NN:          {auc_knn:.4f}")
print("-" * 60)

# Plot ROC curves
plt.figure(figsize=(10, 8))

plt.plot(fpr_rf, tpr_rf, linewidth=2, label=f'Random Forest (AUC = {auc_rf:.3f})')
plt.plot(fpr_svm, tpr_svm, linewidth=2, label=f'SVM (AUC = {auc_svm:.3f})')
plt.plot(fpr_knn, tpr_knn, linewidth=2, label=f'k-NN (AUC = {auc_knn:.3f})')
plt.plot([0, 1], [0, 1], 'k--', linewidth=1, label='Random Classifier')

plt.xlabel('False Positive Rate', fontsize=12, fontweight='bold')
```

```
plt.tight_layout()
plt.savefig('roc_curves.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nROC curves saved: roc_curves.png")

# ============================================================================
# PART 12: FEATURE IMPORTANCE VISUALIZATION
# ============================================================================

print("\n" + "="*80)
print("PART 12: FEATURE IMPORTANCE (RANDOM FOREST)")
print("="*80)

# Plot feature importance
fig, ax = plt.subplots(figsize=(10, 8))

feature_importance_sorted = feature_importance.sort_values('importance')
colors = plt.cm.RdYlGn(np.linspace(0.3, 0.9, len(feature_importance_sorted)))

bars = ax.barh(feature_importance_sorted['feature'],
               feature_importance_sorted['importance'],
               color=colors, edgecolor='black', linewidth=0.5)

ax.set_xlabel('Importance Score', fontsize=12, fontweight='bold')
ax.set_ylabel('Water Quality Parameter', fontsize=12, fontweight='bold')
ax.set_title('Feature Importance (Random Forest)', fontsize=14, fontweight='bold')
ax.grid(True, alpha=0.3, axis='x')

# Add value labels
for bar in bars:
    width = bar.get_width()
    ax.text(width, bar.get_y() + bar.get_height()/2,
            f'{width:.4f}', ha='left', va='center', fontsize=9)

plt.tight_layout()
plt.savefig('feature_importance.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nFeature importance plot saved: feature_importance.png")

# Save feature importance
feature_importance.to_csv('/content/feature_importance.csv', index=False)
print("Feature importance data saved: feature_importance.csv")

# ============================================================================
# PART 13: SAVE ALL RESULTS
# ============================================================================

print("\n" + "="*80)
print("PART 13: SAVING ALL RESULTS")
print("="*80)

# Save all predictions
results_df = pd.DataFrame({
    'true_label': le.inverse_transform(y_test),
    'rf_prediction': le.inverse_transform(y_test_pred_rf),
    'svm_prediction': le.inverse_transform(y_test_pred_svm),
    'knn_prediction': le.inverse_transform(y_test_pred_knn),
    'rf_probability': y_test_proba_rf,
```


```
bars = ax.barh(feature_importance_sorted['feature'],
               feature_importance_sorted['importance'],
               color=colors, edgecolor='black', linewidth=0.5)

ax.set_xlabel('Importance Score', fontsize=12, fontweight='bold')
ax.set_ylabel('Water Quality Parameter', fontsize=12, fontweight='bold')
ax.set_title('Feature Importance (Random Forest)', fontsize=14, fontweight='bold')
ax.grid(True, alpha=0.3, axis='x')

# Add value labels
for bar in bars:
    width = bar.get_width()
    ax.text(width, bar.get_y() + bar.get_height()/2,
            f'{width:.4f}', ha='left', va='center', fontsize=9)

plt.tight_layout()
plt.savefig('feature_importance.png', dpi=300, bbox_inches='tight')
plt.show()

print("\nFeature importance plot saved: feature_importance.png")

# Save feature importance
feature_importance.to_csv('/content/feature_importance.csv', index=False)
print("Feature importance data saved: feature_importance.csv")

# ============================================================================
# PART 13: SAVE ALL RESULTS
# ============================================================================

print("\n" + "="*80)
print("PART 13: SAVING ALL RESULTS")
print("="*80)

# Save all predictions
results_df = pd.DataFrame({
    'true_label': le.inverse_transform(y_test),
    'rf_prediction': le.inverse_transform(y_test_pred_rf),
    'svm_prediction': le.inverse_transform(y_test_pred_svm),
    'knn_prediction': le.inverse_transform(y_test_pred_knn),
    'rf_probability': y_test_proba_rf,
    'svm_probability': y_test_proba_svm,
    'knn_probability': y_test_proba_knn
})
results_df.to_csv('/content/test_predictions.csv', index=False)
print("Test predictions saved: test_predictions.csv")

print("\n" + "="*80)
print("PHASE 4: SUPERVISED CLASSIFICATION COMPLETE!")
print("="*80)

print(f"\nFINAL SUMMARY:")
print(f"  - Total samples: {len(X)}")
print(f"  - Training samples: {len(X_train)}")
print(f"  - Test samples: {len(X_test)}")
print(f"  - Features: {len(feature_columns)}")
print(f"  - Classes: {len(le.classes_)} ({', '.join(le.classes_)})")
print(f"\n  - Model with high accuracy: {best_model_name}")
print(f"  - Best Test Accuracy: {best_model_acc:.2f}%")

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