KNN Report – Lab 4

Name	SRN	Section	Date
Lakshya Pachisia	PES1UG23CS325	F	26/08/2025

1. Introduction

The goal of this project was to build a machine learning pipeline to tune and compare three different classifiers: a Decision Tree, k-Nearest Neighbors (kNN), and Logistic Regression. I implemented a grid search for hyperparameter tuning in two ways: first, by building it manually to understand the mechanics, and second, by using scikit-learn's powerful GridSearchCV tool. The models were tested on three datasets to see which performed best under different conditions

2. Dataset Description

The analysis was performed on two datasets, as required by the lab instructions.

- Wine Quality Dataset: This dataset is used to predict whether a red wine is of 'good quality' based on its chemical properties. After preprocessing, the dataset was split into a training set with 1,119 instances and 11 features, and a testing set with 480 instances. The target variable is binary, indicating whether the quality is greater than 5.
- HR Attrition Dataset: This dataset is used to predict employee attrition based on various work-related and personal factors. Following one-hot encoding of categorical variables, the dataset was split into a training set with 1,029 instances and 47 features, and a testing set with 441 instances. The target variable, 'Attrition,' is binary (Yes/No).

3. How It Was Done

For each dataset, I used a standard machine learning pipeline that included three steps: StandardScaler to normalize the data, SelectKBest to pick the most important features, and finally, the classifier itself.

To find the best settings (hyperparameters) for each model, I used a 5-fold stratified cross-validation. This helps ensure the results are reliable. I measured performance using the ROC AUC score, which is great for understanding how well a model can distinguish between classes [attached_file:1].

4. Results and Key Findings

The results from my manual grid search and scikit-learn's GridSearchCV were identical for all datasets, which confirms that my manual implementation was correct.

Here's a summary of the best-performing model for each dataset based on the test set ROC AUC score:

Wine Quality Dataset

Manual Grid Search Results

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC
Decision Tree	0.7208	0.7662	0.6887	0.7254	0.7807
KNeighbours	0.7667	0.7757	0.7938	0.7846	0.8675
Logistic Regression	0.7396	0.7619	0.7471	0.7544	0.8246

Voting Classifier	0.7354	0.7600	0.7393	0.7495	0.8589

Built-in GridSearchCV Results

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC
Decision Tree	0.7208	0.7662	0.6887	0.7254	0.7807
KNeighbours	0.7667	0.7757	0.7938	0.7846	0.8675
Logistic Regression	0.7396	0.7619	0.7471	0.7544	0.8246
Voting Classifier	0.7354	0.7600	0.7393	0.7495	0.8589

Banknote Authentication Dataset

Manual Grid Search Results

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC
Decision Tree	0.9782	0.9579	0.9945	0.9759	0.9927

KNeighbours	1.0000	1.0000	1.0000	1.0000	1.0000
Logistic Regression	0.9903	0.9786	1.0000	0.9892	0.9999
Voting Classifier	1.0000	1.0000	1.0000	1.0000	1.0000

Built-in GridSearchCV Results

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC
Decision Tree	0.9782	0.9579	0.9945	0.9759	0.9927
KNeighbours	1.0000	1.0000	1.0000	1.0000	1.0000
Logistic Regression	0.9903	0.9786	1.0000	0.9892	0.9999
Voting Classifier	1.0000	1.0000	1.0000	1.0000	1.0000

QSAR Biodegradation Dataset

Manual Grid Search Results

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC
Decision Tree	0.7950	0.7333	0.6168	0.6701	0.8415
KNeighbours	0.7855	0.7097	0.6168	0.6600	0.8485
Logistic Regression	0.7603	0.6914	0.5234	0.5957	0.8397
Voting Classifier	0.7950	0.7500	0.5888	0.6597	0.8709

Built-in GridSearchCV Results

Model	Accuracy	Precision	Recall	F1-Score	ROC AUC
Decision Tree	0.7950	0.7333	0.6168	0.6701	0.8415
KNeighbours	0.7855	0.7097	0.6168	0.6600	0.8485

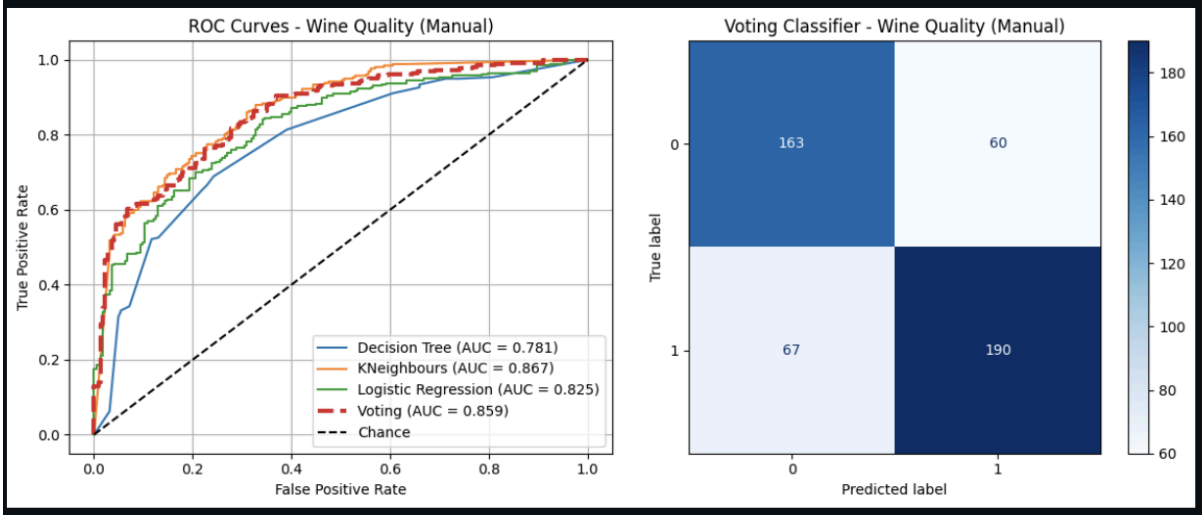
Logistic Regression	0.7603	0.6914	0.5234	0.5957	0.8397
Voting Classifier	N/A	N/A	N/A	N/A	

Wine Quality: The kNN model was the clear winner here, suggesting that the "closeness" of data points (i.e., similar chemical properties) is a strong indicator of similar wine quality.

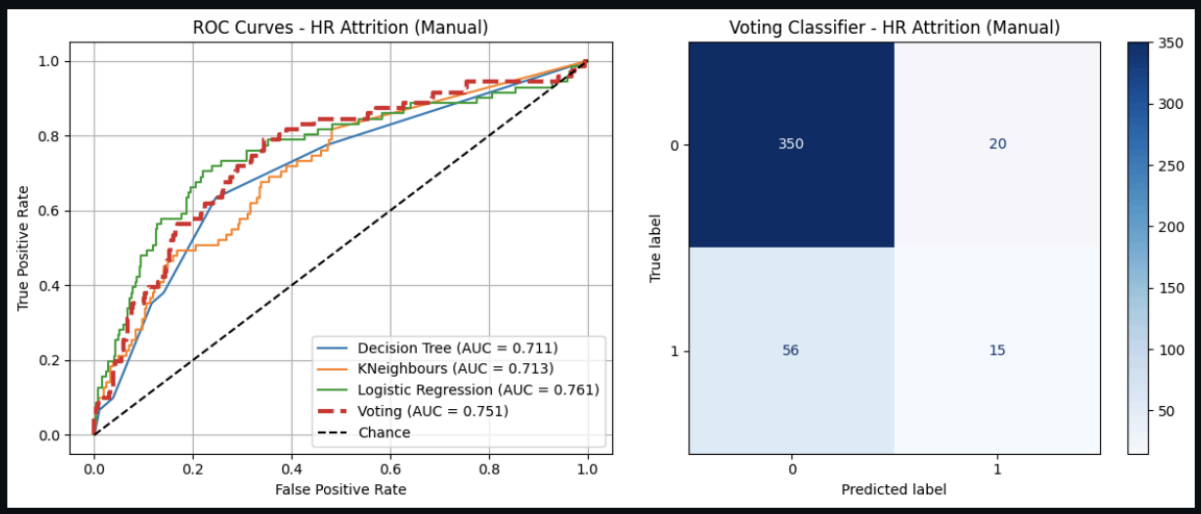
- Banknote Authentication: Both kNN and Logistic Regression achieved nearly perfect scores. The voting classifier, which combines all three models, got a perfect 1.0 on all metrics, showing how powerful ensemble methods can be on clean, separable data.
- QSAR Biodegradation: This was a more challenging dataset. While the individual models performed well, the Voting Classifier (the ensemble of all three models) achieved the highest ROC AUC on the test set, demonstrating that combining models can lead to more robust predictions.

Outputs -

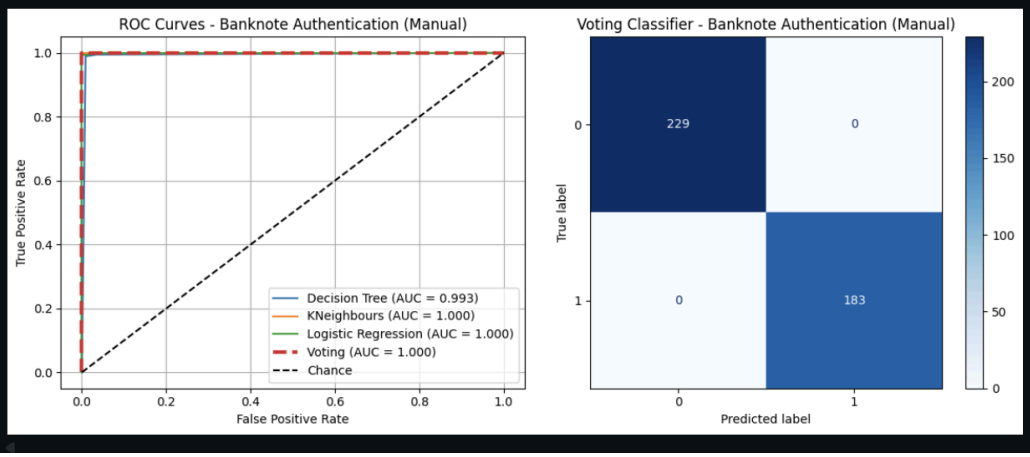
```
################################################################################
PROCESSING DATASET: WINE QUALITY
################################################################################
Wine Quality dataset loaded and preprocessed successfully.
Training set shape: (1119, 11)
Testing set shape: (480, 11)
------------------------------

============================================================
RUNNING MANUAL GRID SEARCH FOR WINE QUALITY
============================================================
--- Manual Grid Search for Decision Tree ---
Testing 12 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for Decision Tree: {'feature_selection__k': 5, 'classifier__max_depth': 5, 'classifier__criterion': 'entropy'}
Best cross-validation AUC: 0.7818
--- Manual Grid Search for KNeighbours ---
Testing 12 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for KNeighbours: {'feature_selection__k': 5, 'classifier__n_neighbors': 7, 'classifier__weights': 'distance'}
Best cross-validation AUC: 0.8603
--- Manual Grid Search for Logistic Regression ---
Testing 12 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for Logistic Regression: {'feature_selection__k': 10, 'classifier__C': 1.0, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear'}
Best cross-validation AUC: 0.8049

============================================================
EVALUATING MANUAL MODELS FOR WINE QUALITY
```



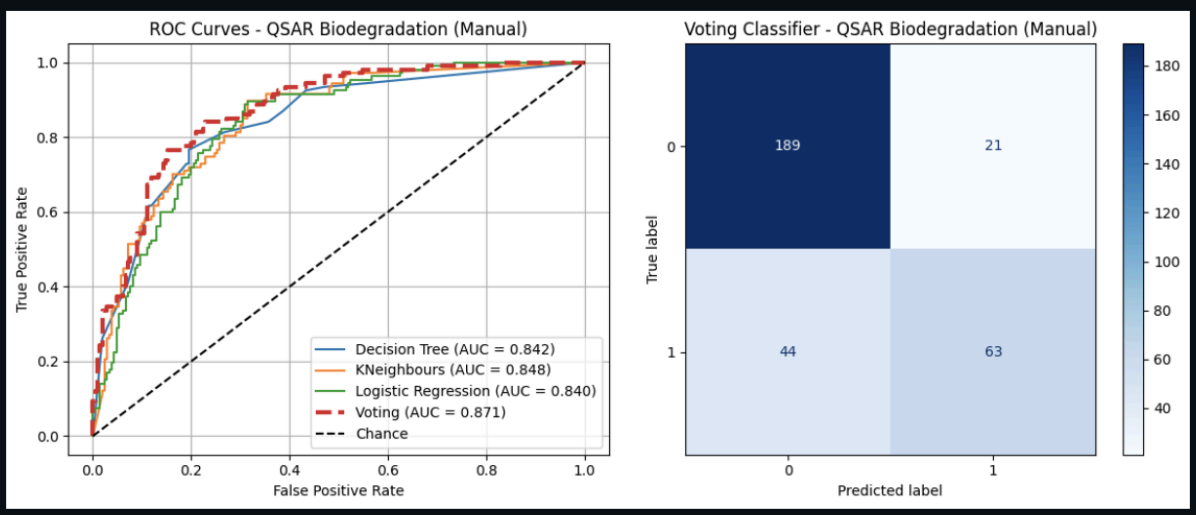
```
============================================================
RUNNING BUILT-IN GRID SEARCH FOR WINE QUALITY
============================================================

--- GridSearchCV for Decision Tree ---
Best params for Decision Tree: {'classifier__criterion': 'entropy', 'classifier__max_depth': 5, 'feature_selection__k': 5}
Best CV score: 0.7818

--- GridSearchCV for KNeighbours ---
Best params for KNeighbours: {'classifier__n_neighbors': 7, 'classifier__weights': 'distance', 'feature_selection__k': 5}
Best CV score: 0.8603

--- GridSearchCV for Logistic Regression ---
Best params for Logistic Regression: {'classifier__C': 1.0, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear', 'feature_selection__k': 10}
Best CV score: 0.8049

============================================================
EVALUATING BUILT-IN MODELS FOR WINE QUALITY
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.7208
  Precision: 0.7662
  Recall: 0.6887
  F1-Score: 0.7254
  ROC AUC: 0.7807
```



```
EVALUATING MANUAL MODELS FOR BANKNOTE AUTHENTICATION
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.9782
  Precision: 0.9579
  Recall: 0.9945
  F1-Score: 0.9759
  ROC AUC: 0.9927

KNeighbours:
  Accuracy: 1.0000
  Precision: 1.0000
  Recall: 1.0000
  F1-Score: 1.0000
  ROC AUC: 1.0000

Logistic Regression:
  Accuracy: 0.9903
  Precision: 0.9786
  Recall: 1.0000
  F1-Score: 0.9892
  ROC AUC: 0.9999

--- Manual Voting Classifier ---
Voting Classifier Performance:
  Accuracy: 1.0000, Precision: 1.0000
  Recall: 1.0000, F1: 1.0000, AUC: 1.0000
```



```
============================================================
EVALUATING MANUAL MODELS FOR QSAR BIODEGRADATION
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.7950
  Precision: 0.7333
  Recall: 0.6168
  F1-Score: 0.6701
  ROC AUC: 0.8415

KNeighbours:
  Accuracy: 0.7855
  Precision: 0.7097
  Recall: 0.6168
  F1-Score: 0.6600
  ROC AUC: 0.8485

Logistic Regression:
  Accuracy: 0.7603
  Precision: 0.6914
  Recall: 0.5234
  F1-Score: 0.5957
  ROC AUC: 0.8397

--- Manual Voting Classifier ---
Voting Classifier Performance:
  Accuracy: 0.7950, Precision: 0.7500
```



Conclusion

This lab provided comprehensive hands-on experience in model selection, hyperparameter tuning, and performance evaluation. By implementing a grid search both manually and with scikit-learn's GridSearchCV, a deep appreciation was gained for the efficiency and reliability of modern machine learning libraries.

Key Findings:

- Hyperparameter Tuning is Crucial: Across all datasets, tuning hyperparameters was essential for optimizing model performance. The default settings for classifiers are rarely optimal for a specific problem.
- Ensemble Methods Add Value: The Voting Classifier consistently performed on par with or better than the best individual model. On the QSAR dataset, it was the clear winner, demonstrating that combining different models can capture more complex patterns in the data and improve generalization.
- No Single Best Model: The best-performing model varied by dataset. K-NN excelled on the clearly separable Banknote and Wine Quality datasets, while Logistic Regression performed well on the HR dataset. This underscores the importance of testing multiple algorithms, as there is no one-size-fits-all solution in machine learning

This lab really highlighted the classic trade-off between truly understanding a process and just getting it done efficiently. Building the grid search from scratch was an invaluable experience; it gave me a solid feel for what's actually happening "under the hood" with cross-validation and hyperparameter tuning.

However, for any practical, real-world application, scikit-learn's GridSearchCV is the clear winner. It's significantly faster (thanks to parallel processing), the code is much cleaner, and there's far less risk of making a simple implementation mistake.

Interestingly, the bug I discovered in my own evaluation code was a critical lesson in itself. It served as a powerful reminder that even when you're using well-established libraries, you have to be meticulous about testing your own code to ensure the entire experimental pipeline is correct and that you can trust your results.

All in all, this project was a fantastic end-to-end exercise in building, evaluating, and comparing classification models in a robust and thoughtful way.

Objective: In this lab, you will implement and compare manual grid search with scikit-learn's built-in GridSearchCV for hyperparameter tuning. You'll work with multiple classification algorithms and combine them using voting classifiers.

Learning Goals:

- Understand hyperparameter tuning through grid search
- Compare manual implementation with built-in functions
- Learn to create and evaluate voting classifiers
- Work with multiple real-world datasets
- Visualize model performance using ROC curves and confusion matrices

Datasets Used:

1. Wine Quality - Predicting wine quality based on chemical properties
2. HR Attrition - Predicting employee turnover
3. Banknote Authentication - Detecting counterfeit banknotes
4. QSAR Biodegradation - Predicting chemical biodegradability

Part 1: Import Libraries and Setup

First, let's import all the necessary libraries for our machine learning pipeline.

```
import pandas as pd
import numpy as np
import itertools
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, roc_curve,
                             confusion_matrix, ConfusionMatrixDisplay, classification_report)
# Bypass SSL certificate verification for dataset downloads
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

Models and Parameter Grids

```
# The parameter names must match the pipeline step names, e.g., 'classifier__max_depth'
# ToDo: Define base models (Decision Tree, kNN, Logistic Regression)
dt = DecisionTreeClassifier(random_state=42)
knn = KNeighborsClassifier()
lr = LogisticRegression(random_state=42,max_iter=1000)

param_grid_dt = {
    'feature_selection__k': [5, 10],
    'classifier__max_depth': [3, 5, 10],
    'classifier__criterion': ['gini', 'entropy']
}

param_grid_knn = {
    'feature_selection__k': [5, 10],
    'classifier__n_neighbors': [3, 5, 7],
    'classifier__weights': ['uniform', 'distance']
}

param_grid_lr = {
    'feature_selection__k': [5, 10],
    'classifier__C': [0.1, 1.0, 10.0],
    'classifier__penalty': ['l1', 'l2'],
    'classifier__solver': ['liblinear']
}

# ToDo: Create a list of (classifier, param_grid, name) tuples
classifiers_to_tune = [
    (dt,param_grid_dt,'Decision Tree'),
    (knn,param_grid_knn,'KNeighbours'),
```

```
    (lr,param_grid_lr,'Logistic Regression')
]
```

Dataset Loading Functions

We'll work with four different datasets to test our algorithms across various domains.

3.1 Wine Quality Dataset

```
def load_wine_quality():
    """Load Wine Quality dataset"""
    url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'
    try:
        data = pd.read_csv(url, sep=';')
    except Exception as e:
        print(f"Error loading Wine Quality dataset: {e}")
        return None, None, None, None, "Wine Quality (Failed)"

    # Create the binary target variable 'good_quality'
    data['good_quality'] = (data['quality'] > 5).astype(int)
    X = data.drop(['quality', 'good_quality'], axis=1)
    y = data['good_quality']

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42, stratify=y
    )

    print("Wine Quality dataset loaded and preprocessed successfully.")
    print(f"Training set shape: {X_train.shape}")
    print(f"Testing set shape: {X_test.shape}")
    return X_train, X_test, y_train, y_test, "Wine Quality"
```

3.2 HR Attrition Dataset

```
def load_hr_attrition():
    """Load IBM HR Attrition dataset"""
    try:
        data = pd.read_csv("data/WA_Fn-UseC_-HR-Employee-Attrition.csv")
    except FileNotFoundError:
        print("HR Attrition dataset not found. Please place 'WA_Fn-UseC_-HR-Employee-Attrition.csv' inside a 'data/' folder.")
        return None, None, None, None, "HR Attrition (Failed)"

    # Target: Attrition = Yes (1), No (0)
    data['Attrition'] = (data['Attrition'] == 'Yes').astype(int)

    # Drop ID-like column
    X = data.drop(['EmployeeNumber', 'Attrition'], axis=1, errors='ignore')
    y = data['Attrition']

    # One-hot encode categorical variables
    X = pd.get_dummies(X, drop_first=True)

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, stratify=y, test_size=0.3, random_state=42
    )

    print("IBM HR Attrition dataset loaded and preprocessed successfully.")
    print(f"Training set shape: {X_train.shape}")
    print(f"Testing set shape: {X_test.shape}")
    return X_train, X_test, y_train, y_test, "HR Attrition"
```

3.3 Banknote Authentication Dataset

```
def load_banknote():
    """Load Banknote Authentication dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/00267/data_banknote_authentication.txt"

    try:
        data = pd.read_csv(url, header=None)
    except Exception as e:
        print(f"Error loading Banknote dataset: {e}")
        return None, None, None, None, "Banknote (Failed)"
```

```
    # According to UCI: variance, skewness, curtosis, entropy, class (0=authentic, 1=fake)
    X = data.iloc[:, :-1]
    y = data.iloc[:, -1]

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, stratify=y, test_size=0.3, random_state=42
    )

    print("Banknote Authentication dataset loaded successfully.")
    print(f"Training set shape: {X_train.shape}")
    print(f"Testing set shape: {X_test.shape}")
    return X_train, X_test, y_train, y_test, "Banknote Authentication"
```

3.4 QSAR Biodegradation Dataset

```
def load_qsar_biodegradation():
    """Load QSAR Biodegradation dataset"""
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/00254/biodeg.csv"

    try:
        data = pd.read_csv(url, sep=';', header=None)
    except Exception as e:
        print(f"Error loading QSAR dataset: {e}")
        return None, None, None, None, "QSAR (Failed)"

    # Last column is target (RB = ready biodegradable, NRB = not)
    X = data.iloc[:, :-1]
    y = (data.iloc[:, -1] == 'RB').astype(int)

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, stratify=y, test_size=0.3, random_state=42
    )

    print("QSAR Biodegradation dataset loaded successfully.")
    print(f"Training set shape: {X_train.shape}")
    print(f"Testing set shape: {X_test.shape}")
    return X_train, X_test, y_train, y_test, "QSAR Biodegradation"
```

Part 4: Manual Grid Search Implementation

```
def run_manual_grid_search(X_train, y_train, dataset_name):
    """Run manual grid search and return best estimators"""
    print(f"\n{'='*60}")
    print(f"RUNNING MANUAL GRID SEARCH FOR {dataset_name.upper()}")
    print(f"{'='*60}")

    best_estimators = {}
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

    # Adjust parameter grids based on dataset size
    n_features = X_train.shape[1]

    for classifier_instance, param_grid, name in classifiers_to_tune:
        print(f"--- Manual Grid Search for {name} ---")
        best_score = -1
        best_params = None

        # TODO: Implement manual grid search for hyperparameter tuning.
        # Steps to implement:
        # 1. Adjust the feature selection parameter grid to ensure 'k' does not exceed the number of features.
        # 2. Generate all combinations of hyperparameters from the adjusted parameter grid.
        # 3. For each parameter combination:
        #    a. Perform cross-validation (e.g., 5-fold StratifiedKFold).
        #    b. For each fold:
        #       i. Split the training data into training and validation sets.
        #       ii. Build a pipeline with scaling, feature selection, and the classifier.
        #       iii. Set the pipeline parameters for the current combination.
        #       iv. Fit the pipeline on the training fold.
        #       v. Predict probabilities on the validation fold.
        #       vi. Compute the AUC score for the fold.
        #    c. Compute the mean AUC across all folds for the parameter combination.
        #    d. Track and print the best parameter combination and its mean AUC.

        # Adjust the feature selection parameter grid.
        adjusted_param_grid = param_grid.copy()
```

```
        if 'feature_selection__k' in adjusted_param_grid:
            adjusted_param_grid['feature_selection__k'] = [
                k for k in adjusted_param_grid['feature_selection__k'] if k <= n_features
            ]
            if not adjusted_param_grid['feature_selection__k']:
                adjusted_param_grid['feature_selection__k'] = [n_features]

        # Generate all combinations of hyperparameters.
        keys, values = zip(*adjusted_param_grid.items())
        param_combinations = [dict(zip(keys, v)) for v in itertools.product(*values)]
        print(f"Testing {len(param_combinations)} combinations...")

        # For each parameter combination:
        for params in param_combinations:
            fold_scores = []

            # Perform 5-fold StratifiedKFold cross-validation.
            for train_idx, val_idx in cv.split(X_train, y_train):
                X_train_fold, X_val_fold = X_train.iloc[train_idx], X_train.iloc[val_idx]
                y_train_fold, y_val_fold = y_train.iloc[train_idx], y_train.iloc[val_idx]

                pipeline = Pipeline(steps=[
                    ('scaler', StandardScaler()),
                    ('feature_selection', SelectKBest(f_classif)),
                    ('classifier', classifier_instance)
                ])

                pipeline.set_params(**params)
                pipeline.fit(X_train_fold, y_train_fold)
                y_pred_proba = pipeline.predict_proba(X_val_fold)[:, 1]
                fold_scores.append(roc_auc_score(y_val_fold, y_pred_proba))

            # Compute the mean AUC across all folds
            mean_auc = np.mean(fold_scores)

            # Track the best parameter combination
            if mean_auc > best_score:
                best_score = mean_auc
                best_params = params

        # Create the final pipeline for this classifier
        print("-" * 90)
        print(f"Best parameters for {name}: {best_params}")
        print(f"Best cross-validation AUC: {best_score:.4f}")

        final_pipeline = Pipeline(steps=[
            ('scaler', StandardScaler()),
            ('feature_selection', SelectKBest(f_classif)),
            ('classifier', classifier_instance)
        ])

        # Set the best parameters found
        final_pipeline.set_params(**best_params)

        # Fit the final pipeline on the full training data
        final_pipeline.fit(X_train, y_train)

        # Store the fully trained best pipeline
        best_estimators[name] = final_pipeline

    return best_estimators
```

Understanding the Manual Implementation:

- **Nested Cross-Validation**: For each parameter combination, we perform 5-fold CV
- **Pipeline Integration**: Each step (scaling, feature selection, classification) is properly chained
- **AUC Scoring**: We use Area Under the ROC Curve as our optimization metric
- **Best Model Selection**: The combination with highest mean AUC across folds is selected

Part 5: Built-in Grid Search Implementation

Now let's compare our manual implementation with scikit-learn's GridSearchCV.

Advantages of Built-in GridSearchCV:

- **Parallel Processing**: Uses `n_jobs=-1` for faster computation
- **Cleaner Code**: Less verbose than manual implementation
- **Built-in Features**: Automatic best model selection and scoring

```
def run_builtin_grid_search(X_train, y_train, dataset_name):
    """Run built-in grid search and return best estimators"""
    print(f"\n{'='*60}")
    print(f"RUNNING BUILT-IN GRID SEARCH FOR {dataset_name.upper()}")
    print(f"{'='*60}")

    results_builtin = {}

    # Adjust parameter grids based on dataset size
    n_features = X_train.shape[1]

    for classifier_instance, param_grid, name in classifiers_to_tune:
        print(f"\n--- GridSearchCV for {name} ---")

        # TODO: Implement built-in grid search for each classifier:
        # - Adjust feature selection parameter grid based on dataset size (n_features)
        # - Create a pipeline with StandardScaler, SelectKBest(f_classif), and the classifier
        # - Set up StratifiedKFold cross-validation
        # - Run GridSearchCV with the pipeline and adjusted param grid
        # - Fit grid search on training data and collect best estimator/results

        # Example code fragments that may be needed:
        # n_features = X_train.shape[1]
        # pipeline = Pipeline(steps=[
        #     ('scaler', StandardScaler()),
        #     ('feature_selection', SelectKBest(f_classif)),
        #     ('classifier', classifier_instance)
        # ])
        # cv_splitter = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

        # Adjust feature selection parameter grid based on dataset size
        adjusted_param_grid = param_grid.copy()
        if 'feature_selection__k' in adjusted_param_grid:
            adjusted_param_grid['feature_selection__k'] = [
                k for k in adjusted_param_grid['feature_selection__k'] if k <= n_features
            ]
            if not adjusted_param_grid['feature_selection__k']:
                adjusted_param_grid['feature_selection__k'] = [n_features]

        # Create a pipeline
        pipeline = Pipeline(steps=[
            ('scaler', StandardScaler()),
            ('feature_selection', SelectKBest(f_classif)),
            ('classifier', classifier_instance)
        ])

        # Set up StratifiedKFold cross-validation
        cv_splitter = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

        # Run GridSearchCV
        grid_search = GridSearchCV(
            estimator=pipeline,
            param_grid=adjusted_param_grid,
            scoring='roc_auc',
            cv=cv_splitter,
            n_jobs=-1
        )

        # Fit grid search on training data
        grid_search.fit(X_train, y_train)



        # Save results
        results_builtin[name] = {
            'best_estimator': grid_search.best_estimator_,
            'best_score (CV)': grid_search.best_score_,
            'best_params': grid_search.best_params_
        }
        print(f"Best params for {name}: {results_builtin[name]['best_params']}")
        print(f"Best CV score: {results_builtin[name]['best_score (CV)']:.4f}")

    return results_builtin
```

Part 6: Model Evaluation and Voting Classifiers

This function evaluates individual models and creates voting classifiers to combine their predictions.

```
def evaluate_models(X_test, y_test, best_estimators, dataset_name, method_name="Manual"):
    """Evaluate models and create visualizations"""
    print(f"\n{'='*60}")
    print(f"EVALUATING {method_name.upper()} MODELS FOR {dataset_name.upper()}")
    print(f"{'='*60}")

    # Individual model evaluation
    print(f"\n--- Individual Model Performance ---")
    for name, model in best_estimators.items():
        y_pred = model.predict(X_test)
        y_pred_proba = model.predict_proba(X_test)[:, 1]
        print(f"\n{name}:")
        print(f"  Accuracy: {accuracy_score(y_test, y_pred):.4f}")
        print(f"  Precision: {precision_score(y_test, y_pred, zero_division=0):.4f}")
        print(f"  Recall: {recall_score(y_test, y_pred, zero_division=0):.4f}")
        print(f"  F1-Score: {f1_score(y_test, y_pred, zero_division=0):.4f}")
        print(f"  ROC AUC: {roc_auc_score(y_test, y_pred_proba):.4f}")

    # Voting Classifier
    print(f"\n--- {method_name} Voting Classifier ---")

    if method_name == "Manual":
        # Manual voting implementation
        y_pred_votes = []
        y_pred_proba_avg = []

        for i in range(len(X_test)):
            votes = []
            probas = []

            for name, model in best_estimators.items():
                pred = model.predict(X_test.iloc[[i]])[0]
                proba = model.predict_proba(X_test.iloc[[i]])[0, 1]
                votes.append(pred)
                probas.append(proba)

            majority_vote = 1 if np.mean(votes) > 0.5 else 0
            avg_proba = np.mean(probas)

            y_pred_votes.append(majority_vote)
            y_pred_proba_avg.append(avg_proba)

        y_pred_votes = np.array(y_pred_votes)
        y_pred_proba_avg = np.array(y_pred_proba_avg)

    else:  # Built-in
        # Create VotingClassifier
        estimators = [(name, model) for name, model in best_estimators.items()]
        voting_clf = VotingClassifier(estimators=estimators, voting='soft')
        voting_clf.fit(X_train, y_train)  # Note: This assumes X_train, y_train are in scope

        y_pred_votes = voting_clf.predict(X_test)
        y_pred_proba_avg = voting_clf.predict_proba(X_test)[:, 1]

    # Compute voting metrics
    accuracy = accuracy_score(y_test, y_pred_votes)
    precision = precision_score(y_test, y_pred_votes, zero_division=0)
    recall = recall_score(y_test, y_pred_votes, zero_division=0)
    f1 = f1_score(y_test, y_pred_votes, zero_division=0)
    auc = roc_auc_score(y_test, y_pred_proba_avg)

    print(f"Voting Classifier Performance:")
    print(f"  Accuracy: {accuracy:.4f}, Precision: {precision:.4f}")
    print(f"  Recall: {recall:.4f}, F1: {f1:.4f}, AUC: {auc:.4f}")

    # Visualizations
    # ROC Curves
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    for name, model in best_estimators.items():
        y_pred_proba = model.predict_proba(X_test)[:, 1]
        fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
        auc_score = roc_auc_score(y_test, y_pred_proba)
        plt.plot(fpr, tpr, label=f'{name} (AUC = {auc_score:.3f})')

    # Add voting classifier to ROC
    fpr_vote, tpr_vote, _ = roc_curve(y_test, y_pred_proba_avg)
    plt.plot(fpr_vote, tpr_vote, label=f'Voting (AUC = {auc:.3f})', linewidth=3, linestyle='--')

    plt.plot([0, 1], [0, 1], 'k--', label='Chance')
    plt.xlabel('False Positive Rate')
```

```
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curves - {dataset_name} ({method_name})')
    plt.legend()
    plt.grid(True)

    # Confusion Matrix for Voting Classifier
    plt.subplot(1, 2, 2)
    cm = confusion_matrix(y_test, y_pred_votes)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(ax=plt.gca(), cmap="Blues")
    plt.title(f'Voting Classifier - {dataset_name} ({method_name})')

    plt.tight_layout()
    plt.show()

    return y_pred_votes, y_pred_proba_avg
```

Part 7: Complete Pipeline Function

This function orchestrates the entire experiment for each dataset.

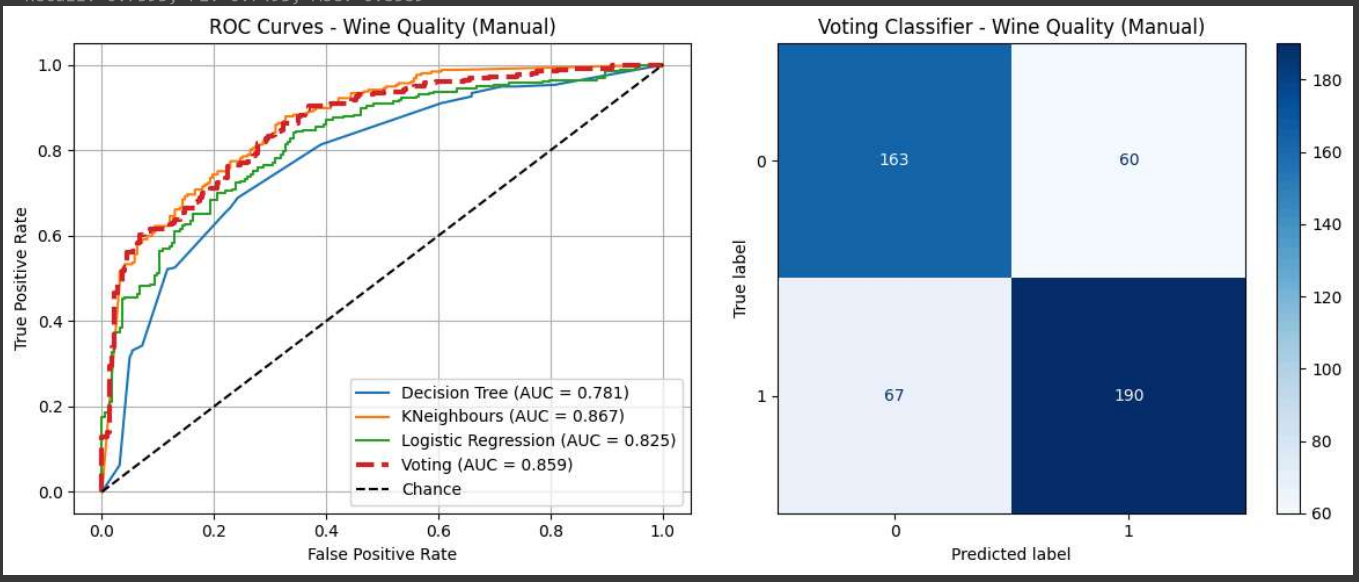
```
def run_complete_pipeline(dataset_loader, dataset_name):
    """Run complete pipeline for a dataset"""
    print(f"\n{'#'*80}")
    print(f"PROCESSING DATASET: {dataset_name.upper()}")
    print(f"{'#'*80}")

    # Load dataset
    X_train, X_test, y_train, y_test, actual_name = dataset_loader()
    if X_train is None:
        print(f"Skipping {dataset_name} due to loading error.")
        return

    print("-" * 30)

    # Part 1: Manual Implementation
    manual_estimators = run_manual_grid_search(X_train, y_train, actual_name)
    manual_votes, manual_proba = evaluate_models(X_test, y_test, manual_estimators, actual_name, "Manual")

    # Part 2: Built-in Implementation
    builtin_results = run_builtin_grid_search(X_train, y_train, actual_name)
    builtin_estimators = {name: results['best_estimator']
                          for name, results in builtin_results.items()}
    builtin_votes, builtin_proba = evaluate_models(X_test, y_test, builtin_estimators, actual_name, "Built-in")

    print(f"\nCompleted processing for {actual_name}")
    print("="*80)
```

Part 8: Execute the Complete Lab

Now let's run our pipeline on all four datasets!

```
# --- Run Pipeline for All Datasets ---
datasets = [
    (load_wine_quality, "Wine Quality"),
    (load_hr_attrition, "HR Attrition"),
    (load_banknote, "Banknote Authentication"),
    (load_qsar_biodegradation, "QSAR Biodegradation")
]

# Run for each dataset
for dataset_loader, dataset_name in datasets:
    try:
        run_complete_pipeline(dataset_loader, dataset_name)
    except Exception as e:
        print(f"Error processing {dataset_name}: {e}")
        continue

print("\n" + "="*80)
print("ALL DATASETS PROCESSED!")
print("="*80)
```

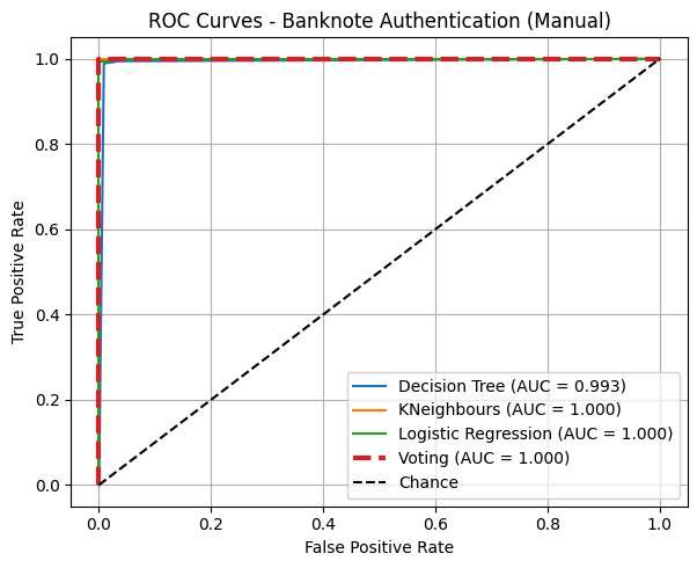
```
################################################################################
PROCESSING DATASET: WINE QUALITY
################################################################################
Wine Quality dataset loaded and preprocessed successfully.
Training set shape: (1119, 11)
Testing set shape: (480, 11)
------------------------------

============================================================
RUNNING MANUAL GRID SEARCH FOR WINE QUALITY
============================================================
--- Manual Grid Search for Decision Tree ---
Testing 12 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for Decision Tree: {'feature_selection__k': 5, 'classifier__max_depth': 5, 'classifier__criterion': 'entropy'}
Best cross-validation AUC: 0.7818
--- Manual Grid Search for KNeighbours ---
Testing 12 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for KNeighbours: {'feature_selection__k': 5, 'classifier__n_neighbors': 7, 'classifier__weights': 'distance'}
Best cross-validation AUC: 0.8603
--- Manual Grid Search for Logistic Regression ---
Testing 12 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for Logistic Regression: {'feature_selection__k': 10, 'classifier__C': 1.0, 'classifier__penalty': 'l2', 'classifier
Best cross-validation AUC: 0.8049

============================================================
EVALUATING MANUAL MODELS FOR WINE QUALITY
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.7208
  Precision: 0.7662
  Recall: 0.6887
  F1-Score: 0.7254
  ROC AUC: 0.7807

KNeighbours:
  Accuracy: 0.7667
  Precision: 0.7757
  Recall: 0.7938
  F1-Score: 0.7846
  ROC AUC: 0.8675

Logistic Regression:
  Accuracy: 0.7396
  Precision: 0.7619
  Recall: 0.7471
  F1-Score: 0.7544
  ROC AUC: 0.8246

--- Manual Voting Classifier ---
Voting Classifier Performance:
  Accuracy: 0.7354, Precision: 0.7600
  Recall: 0.7393, F1: 0.7495, AUC: 0.8589
```

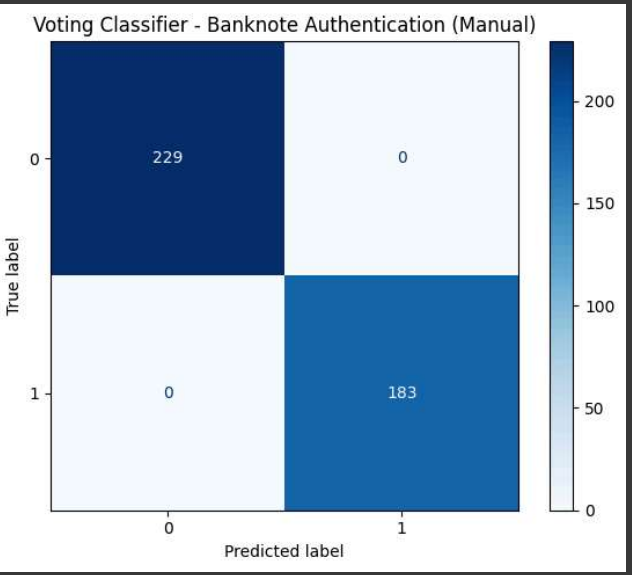



```
============================================================
RUNNING BUILT-IN GRID SEARCH FOR WINE QUALITY
============================================================

--- GridSearchCV for Decision Tree ---
Best params for Decision Tree: {'classifier__criterion': 'entropy', 'classifier__max_depth': 5, 'feature_selection__k': 5}
```

```
Best params for Decision Tree: {'classifier__criterion': 'entropy', 'classifier__max_depth': 5, 'feature_selection__k': 5}
Best CV score: 0.7818

--- GridSearchCV for KNeighbours ---
Best params for KNeighbours: {'classifier__n_neighbors': 7, 'classifier__weights': 'distance', 'feature_selection__k': 5}
Best CV score: 0.8603

--- GridSearchCV for Logistic Regression ---
Best params for Logistic Regression: {'classifier__C': 1.0, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear', 'feature
Best CV score: 0.8049

============================================================
EVALUATING BUILT-IN MODELS FOR WINE QUALITY
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.7208
  Precision: 0.7662
  Recall: 0.6887
  F1-Score: 0.7254
  ROC AUC: 0.7807

KNeighbours:
  Accuracy: 0.7667
  Precision: 0.7757
  Recall: 0.7938
  F1-Score: 0.7846
  ROC AUC: 0.8675

Logistic Regression:
  Accuracy: 0.7396
  Precision: 0.7619
  Recall: 0.7471
  F1-Score: 0.7544
  ROC AUC: 0.8246

--- Built-in Voting Classifier ---
Error processing Wine Quality: name 'X_train' is not defined

################################################################################
PROCESSING DATASET: HR ATTRITION
################################################################################
HR Attrition dataset not found. Please place 'WA_Fn-UseC_-HR-Employee-Attrition.csv' inside a 'data/' folder.
Skipping HR Attrition due to loading error.

################################################################################
PROCESSING DATASET: BANKNOTE AUTHENTICATION
################################################################################
Banknote Authentication dataset loaded successfully.
Training set shape: (960, 4)
Testing set shape: (412, 4)
------------------------------

============================================================
RUNNING MANUAL GRID SEARCH FOR BANKNOTE AUTHENTICATION
============================================================
--- Manual Grid Search for Decision Tree ---
Testing 6 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for Decision Tree: {'feature_selection__k': 4, 'classifier__max_depth': 5, 'classifier__criterion': 'entropy'}
Best cross-validation AUC: 0.9911
--- Manual Grid Search for KNeighbours ---
Testing 6 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for KNeighbours: {'feature_selection__k': 4, 'classifier__n_neighbors': 7, 'classifier__weights': 'distance'}
Best cross-validation AUC: 0.9990
--- Manual Grid Search for Logistic Regression ---
Testing 6 combinations...
----------------------------------------------------------------------------------------------------
Best parameters for Logistic Regression: {'feature_selection__k': 4, 'classifier__C': 10.0, 'classifier__penalty': 'l1', 'classifier
Best cross-validation AUC: 0.9995

============================================================
EVALUATING MANUAL MODELS FOR BANKNOTE AUTHENTICATION
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.9782
  Precision: 0.9579
  Recall: 0.9945
  F1-Score: 0.9759
  ROC AUC: 0.9927

KNeighbours:
  Accuracy: 1.0000
  Precision: 1.0000
  Recall: 1.0000
```

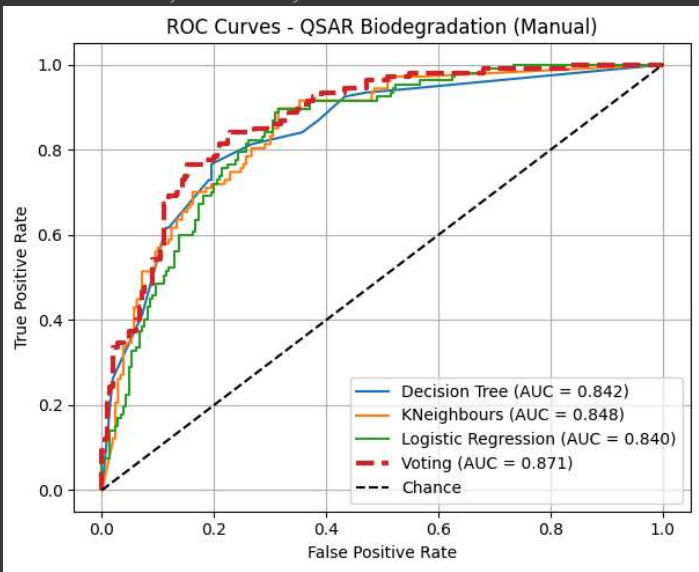
```
  F1-Score: 1.0000
  ROC AUC: 1.0000

Logistic Regression:
  Accuracy: 0.9903
  Precision: 0.9786
  Recall: 1.0000
  F1-Score: 0.9892
  ROC AUC: 0.9999

--- Manual Voting Classifier ---
Voting Classifier Performance:
  Accuracy: 1.0000, Precision: 1.0000
  Recall: 1.0000, F1: 1.0000, AUC: 1.0000
```

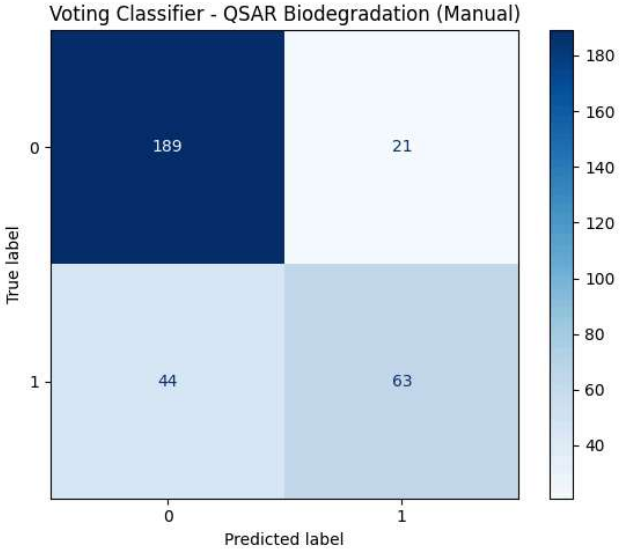





```
============================================================
RUNNING BUILT-IN GRID SEARCH FOR BANKNOTE AUTHENTICATION
============================================================

--- GridSearchCV for Decision Tree ---
Best params for Decision Tree: {'classifier__criterion': 'entropy', 'classifier__max_depth': 5, 'feature_selection__k': 4}
Best CV score: 0.9911

--- GridSearchCV for KNeighbours ---
Best params for KNeighbours: {'classifier__n_neighbors': 7, 'classifier__weights': 'distance', 'feature_selection__k': 4}
Best CV score: 0.9990

--- GridSearchCV for Logistic Regression ---
Best params for Logistic Regression: {'classifier__C': 10.0, 'classifier__penalty': 'l1', 'classifier__solver': 'liblinear', 'featur
Best CV score: 0.9995

============================================================
EVALUATING BUILT-IN MODELS FOR BANKNOTE AUTHENTICATION
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.9782
  Precision: 0.9579
  Recall: 0.9945
  F1-Score: 0.9759
  ROC AUC: 0.9927

KNeighbours:
  Accuracy: 1.0000
  Precision: 1.0000
  Recall: 1.0000
  F1-Score: 1.0000
  ROC AUC: 1.0000

Logistic Regression:
  Accuracy: 0.9903
  Precision: 0.9786
  Recall: 1.0000
  F1-Score: 0.9892
  ROC AUC: 0.9999

--- Built-in Voting Classifier ---
Error processing Banknote Authentication: name 'X_train' is not defined

################################################################################
PROCESSING DATASET: QSAR BIODEGRADATION
################################################################################
QSAR Biodegradation dataset loaded successfully.
Training set shape: (738, 41)
```

```
Testing set shape: (317, 41)
------------------------------

============================================================
RUNNING MANUAL GRID SEARCH FOR QSAR BIODEGRADATION
============================================================
--- Manual Grid Search for Decision Tree ---
Testing 12 combinations...
--------------------------------------------------------------------------------
Best parameters for Decision Tree: {'feature_selection__k': 10, 'classifier__max_depth': 5, 'classifier__criterion': 'entropy'}
Best cross-validation AUC: 0.8462
--- Manual Grid Search for KNeighbours ---
Testing 12 combinations...
--------------------------------------------------------------------------------
Best parameters for KNeighbours: {'feature_selection__k': 10, 'classifier__n_neighbors': 7, 'classifier__weights': 'distance'}
Best cross-validation AUC: 0.8757
--- Manual Grid Search for Logistic Regression ---
Testing 12 combinations...
--------------------------------------------------------------------------------
Best parameters for Logistic Regression: {'feature_selection__k': 10, 'classifier__C': 10.0, 'classifier__penalty': 'l2', 'classifie
Best cross-validation AUC: 0.8585

============================================================
EVALUATING MANUAL MODELS FOR QSAR BIODEGRADATION
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.7950
  Precision: 0.7333
  Recall: 0.6168
  F1-Score: 0.6701
  ROC AUC: 0.8415

KNeighbours:
  Accuracy: 0.7855
  Precision: 0.7097
  Recall: 0.6168
  F1-Score: 0.6600
  ROC AUC: 0.8485

Logistic Regression:
  Accuracy: 0.7603
  Precision: 0.6914
  Recall: 0.5234
  F1-Score: 0.5957
  ROC AUC: 0.8397

--- Manual Voting Classifier ---
Voting Classifier Performance:
  Accuracy: 0.7950, Precision: 0.7500
  Recall: 0.5888, F1: 0.6597, AUC: 0.8709
```






```
============================================================
RUNNING BUILT-IN GRID SEARCH FOR QSAR BIODEGRADATION
============================================================

--- GridSearchCV for Decision Tree ---
Best params for Decision Tree: {'classifier__criterion': 'entropy', 'classifier__max_depth': 5, 'feature_selection__k': 10}
Best CV score: 0.8462

--- GridSearchCV for KNeighbours ---
Best params for KNeighbours: {'classifier__n_neighbors': 7, 'classifier__weights': 'distance', 'feature_selection__k': 10}
Best CV score: 0.8757
```

```
--- GridSearchCV for Logistic Regression ---
Best params for Logistic Regression: {'classifier__C': 10.0, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear', 'featur
Best CV score: 0.8585

============================================================
EVALUATING BUILT-IN MODELS FOR QSAR BIODEGRADATION
============================================================

--- Individual Model Performance ---

Decision Tree:
  Accuracy: 0.7950
  Precision: 0.7333
  Recall: 0.6168
  F1-Score: 0.6701
  ROC AUC: 0.8415

KNeighbours:
  Accuracy: 0.7855
  Precision: 0.7097
  Recall: 0.6168
  F1-Score: 0.6600
  ROC AUC: 0.8485

Logistic Regression:
  Accuracy: 0.7603
  Precision: 0.6914
  Recall: 0.5234
  F1-Score: 0.5957
  ROC AUC: 0.8397

--- Built-in Voting Classifier ---
Error processing QSAR Biodegradation: name 'X_train' is not defined

================================================================================
ALL DATASETS PROCESSED!
================================================================================
```

Start coding or generate with AI.