Group Project 1 - Diabetes

By Charles Dilger and Ahsan Imran

[ ] ↳ 43 cells hidden

Comprehensive EDA is now complete. The data is balanced and ready for modeling.

Logistic Regression

We will now split the data into training and testing, using a stratified split since the model is slightly unbalanced. We will run a logistic regression model on the data, and continue by evaluating the metrics.

```
# Train/Test Split and Logistic Regression Model
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import pandas as pd
import numpy as np

print("TRAIN/TEST SPLIT AND LOGISTIC REGRESSION")
print("=" * 45)

# Prepare features and target
feature_cols = ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFunction_lo
X = df_model_ready[feature_cols]
y = df_model_ready['Outcome']

print(f"Dataset shape: {X.shape}")
print(f"Features: {feature_cols}")

# Stratified train/test split (80/20)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y  # Ensures both sets have same class distribution
)

print(f"\nTrain/Test Split:")
```

```
print(f"Training set: {X_train.shape[0]} samples")
print(f"Test set: {X_test.shape[0]} samples")

# Check class distributions
print(f"\nClass distribution in training set:")
print(y_train.value_counts(normalize=True).sort_index())
print(f"\nClass distribution in test set:")
print(y_test.value_counts(normalize=True).sort_index())

# Create logistic regression model
print(f"\nCreating Logistic Regression Model:")
logit_model = LogisticRegression(random_state=42)
logit_model.fit(X_train, y_train)

# Generate predictions (classification vector)
y_pred = logit_model.predict(X_test)
y_pred_proba = logit_model.predict_proba(X_test)[:, 1]  # Probability of class 1

print(f"Model trained successfully")
print(f"Classification vector (first 10): {y_pred[:10]}")
print(f"Prediction probabilities (first 10): {y_pred_proba[:10]}")

print(f"\nLogistic Regression coefficients:")
for i, feature in enumerate(feature_cols):
    coef = logit_model.coef_[0][i]
    print(f"  {feature}: {coef:.4f}")

print(f"Intercept: {logit_model.intercept_[0]:.4f}")

print(f"\nReady for performance evaluation metrics")
```

```
TRAIN/TEST SPLIT AND LOGISTIC REGRESSION
==========================================
Dataset shape: (768, 3)
Features: ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFunction_log

Train/Test Split:
Training set: 614 samples
Test set: 154 samples

Class distribution in training set:
Outcome
0    0.651466
1    0.348534
Name: proportion, dtype: float64

Class distribution in test set:
Outcome
0    0.649351
1    0.350649
Name: proportion, dtype: float64

Creating Logistic Regression Model:
```

```
Creating Logistic Regression model...
Model trained successfully
Classification vector (first 10): [1 0 0 0 0 0 0 1 0 1]
Prediction probabilities (first 10): [0.70659351 0.05780428 0.30261044 0.37234667 0.0
 0.1972052  0.96816478 0.13078798 0.56221514]

Logistic Regression coefficients:
  Glucose_scaled: 1.2784
  Pregnancies_sqrt_scaled: 0.3108
  DiabetesPedigreeFunction_log_scaled: 0.3325
Intercept: -0.8079

Ready for performance evaluation metrics
```

The model is performing as expected. Let's write our own code for Precision, Accuracy, Recall, Specificity, and F1 score

```
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, conf
import pandas as pd

def custom_confusion_matrix(y_true, y_pred):
    """Calculate confusion matrix components manually"""
    TP = sum((y_true == 1) & (y_pred == 1))
    FP = sum((y_true == 0) & (y_pred == 1))
    TN = sum((y_true == 0) & (y_pred == 0))
    FN = sum((y_true == 1) & (y_pred == 0))
    return TP, FP, TN, FN

def custom_accuracy(y_true, y_pred):
    """Accuracy = (TP + TN) / (TP + TN + FP + FN)"""
    TP, FP, TN, FN = custom_confusion_matrix(y_true, y_pred)
    return (TP + TN) / (TP + TN + FP + FN)

def custom_precision(y_true, y_pred):
    """Precision = TP / (TP + FP) - Of all predicted positive, how many are actually posi
    TP, FP, TN, FN = custom_confusion_matrix(y_true, y_pred)
    if (TP + FP) == 0:
        return 0
    return TP / (TP + FP)

def custom_recall(y_true, y_pred):
    """Recall (Sensitivity) = TP / (TP + FN) - Of all actual positive, how many did we pr
    TP, FP, TN, FN = custom_confusion_matrix(y_true, y_pred)
    if (TP + FN) == 0:
        return 0
    return TP / (TP + FN)

def custom_specificity(y_true, y_pred):
    """Specificity = TN / (TN + FP) - Of all actual negative, how many did we predict cor
    TP, FP, TN, FN = custom_confusion_matrix(y_true, y_pred)
```

```
    TP, FP, TN, FN = custom_confusion_matrix(y_true, y_pred)
    if (TN + FP) == 0:
        return 0
    return TN / (TN + FP)

def custom_f1_score(y_true, y_pred):
    """F1 Score = 2 * (Precision * Recall) / (Precision + Recall)"""
    precision = custom_precision(y_true, y_pred)
    recall = custom_recall(y_true, y_pred)
    if (precision + recall) == 0:
        return 0
    return 2 * (precision * recall) / (precision + recall)

    # Test our custom functions against sklearn
print("CUSTOM PERFORMANCE METRICS EVALUATION")
print("=" * 45)

# Calculate custom metrics
TP, FP, TN, FN = custom_confusion_matrix(y_test, y_pred)
custom_acc = custom_accuracy(y_test, y_pred)
custom_prec = custom_precision(y_test, y_pred)
custom_rec = custom_recall(y_test, y_pred)
custom_spec = custom_specificity(y_test, y_pred)
custom_f1 = custom_f1_score(y_test, y_pred)

# Calculate sklearn metrics for comparison
sklearn_acc = accuracy_score(y_test, y_pred)
sklearn_prec = precision_score(y_test, y_pred)
sklearn_rec = recall_score(y_test, y_pred)
sklearn_f1 = f1_score(y_test, y_pred)

# Display confusion matrix components
print("Confusion Matrix Components:")
print(f"True Positives (TP):  {TP}")
print(f"False Positives (FP): {FP}")
print(f"True Negatives (TN):  {TN}")
print(f"False Negatives (FN): {FN}")

print(f"\nCustom vs Sklearn Comparison:")
print(f"{'Metric':<12} {'Custom':<8} {'Sklearn':<8} {'Match':<5}")
print("-" * 35)
print(f"{'Accuracy':<12} {custom_acc:.4f}   {sklearn_acc:.4f}   {abs(custom_acc - sklearn
print(f"{'Precision':<12} {custom_prec:.4f}   {sklearn_prec:.4f}   {abs(custom_prec - skl
print(f"{'Recall':<12} {custom_rec:.4f}   {sklearn_rec:.4f}   {abs(custom_rec - sklearn_r
print(f"{'Specificity':<12} {custom_spec:.4f}   {'N/A':<8} {'N/A'}")
print(f"{'F1-Score':<12} {custom_f1:.4f}   {sklearn_f1:.4f}   {abs(custom_f1 - sklearn_f1

print(f"\nPerformance Summary:")
print(f"Model correctly classified {custom_acc:.1%} of test cases")
print(f"Of predicted diabetes cases, {custom_prec:.1%} were correct (Precision)")
print(f"Of actual diabetes cases, {custom_rec:.1%} were detected (Recall/Sensitivity)")
```

```
print(f"Of actual non-diabetes cases, {custom_spec:.1%} were correctly identified (Specif

# Verification: All custom functions match sklearn
all_match = all([
    abs(custom_acc - sklearn_acc) < 1e-10,
    abs(custom_prec - sklearn_prec) < 1e-10,
    abs(custom_rec - sklearn_rec) < 1e-10,
    abs(custom_f1 - sklearn_f1) < 1e-10
])

print(f"\nVerification: {'All custom functions match sklearn exactly!' if all_match else
```

```
CUSTOM PERFORMANCE METRICS EVALUATION
==========================================
Confusion Matrix Components:
True Positives (TP):  26
False Positives (FP): 17
True Negatives (TN):  83
False Negatives (FN): 28

Custom vs Sklearn Comparison:
Metric       Custom   Sklearn  Match
-----------------------------------
Accuracy     0.7078   0.7078   True
Precision    0.6047   0.6047   True
Recall       0.4815   0.4815   True
Specificity  0.8300   N/A      N/A
F1-Score     0.5361   0.5361   True

Performance Summary:
Model correctly classified 70.8% of test cases
Of predicted diabetes cases, 60.5% were correct (Precision)
Of actual diabetes cases, 48.1% were detected (Recall/Sensitivity)
Of actual non-diabetes cases, 83.0% were correctly identified (Specificity)

Verification: All custom functions match sklearn exactly!
```

Looks good, but we aren't done. Let's try the same, but with different hyperparameters.

```
# Logistic Regression Hyperparameter Tuning and Documentation
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
import pandas as pd
import numpy as np

print("LOGISTIC REGRESSION HYPERPARAMETER TUNING")
print("=" * 50)

# Define hyperparameter combinations to test
hyperparameters = [
    # Test regularization strength (C)
```

```
    {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs', 'class_weight': None, 'description':
    {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs', 'class_weight': None, 'description': '
    {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs', 'class_weight': None, 'description': '
    {'C': 10.0, 'penalty': 'l2', 'solver': 'lbfgs', 'class_weight': None, 'description':
    {'C': 100.0, 'penalty': 'l2', 'solver': 'lbfgs', 'class_weight': None, 'description':

    # Test class balancing
    {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs', 'class_weight': 'balanced', 'descripti

    # Test different penalties
    {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear', 'class_weight': None, 'description
    {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear', 'class_weight': 'balanced', 'descr

    # Test no regularization
    {'C': 1.0, 'penalty': None, 'solver': 'lbfgs', 'class_weight': None, 'description': '
    {'C': 1.0, 'penalty': None, 'solver': 'lbfgs', 'class_weight': 'balanced', 'descripti
]

# Store results for comparison
results = []

print("Testing hyperparameter combinations with 5-fold cross-validation:")
print("-" * 70)

for i, params in enumerate(hyperparameters):
    # Extract description and remove from params for model
    description = params.pop('description')

    # Create and evaluate model
    model = LogisticRegression(random_state=42, max_iter=1000, **params)

    # 5-fold cross-validation scores
    cv_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')
    cv_mean = cv_scores.mean()
    cv_std = cv_scores.std()

    # Fit model and get test performance
    model.fit(X_train, y_train)
    y_pred_hp = model.predict(X_test)
    test_accuracy = custom_accuracy(y_test, y_pred_hp)
    test_precision = custom_precision(y_test, y_pred_hp)
    test_recall = custom_recall(y_test, y_pred_hp)
    test_f1 = custom_f1_score(y_test, y_pred_hp)

    # Store results
    result = {
        'Config': i+1,
        'Description': description,
        'CV_Mean': cv_mean,
        'CV_Std': cv_std,
        'Test_Accuracy': test_accuracy
```

```
            'Test_Accuracy': test_accuracy,
            'Test_Precision': test_precision,
            'Test_Recall': test_recall,
            'Test_F1': test_f1,
            'Parameters': str(params)
        }
        results.append(result)

        print(f"Config {i+1}: {description}")
        print(f"  CV Score: {cv_mean:.4f} (±{cv_std:.4f})")
        print(f"  Test - Acc: {test_accuracy:.4f}, Prec: {test_precision:.4f}, Rec: {test_rec
        print()

    # Create results dataframe and find best performing models
    results_df = pd.DataFrame(results)

    print("=" * 70)
    print("HYPERPARAMETER TUNING RESULTS SUMMARY")
    print("=" * 70)

    # Best by different metrics
    best_cv = results_df.loc[results_df['CV_Mean'].idxmax()]
    best_test_acc = results_df.loc[results_df['Test_Accuracy'].idxmax()]
    best_f1 = results_df.loc[results_df['Test_F1'].idxmax()]
    best_recall = results_df.loc[results_df['Test_Recall'].idxmax()]

    print("Best performing configurations:")
    print(f"Best CV Score: Config {best_cv['Config']} - {best_cv['Description']} (CV: {best_c
    print(f"Best Test Accuracy: Config {best_test_acc['Config']} - {best_test_acc['Descriptio
    print(f"Best F1 Score: Config {best_f1['Config']} - {best_f1['Description']} (F1: {best_f
    print(f"Best Recall: Config {best_recall['Config']} - {best_recall['Description']} (Recal

    print(f"\nKey Findings:")
    print(f"1. Regularization impact: Compare configs 1-5 (C values 0.01 to 100)")
    print(f"2. Class balancing impact: Compare configs with/without 'balanced' class_weight")
    print(f"3. Penalty type impact: Compare L1 vs L2 vs None regularization")

    # Select best overall model (highest F1 score for imbalanced data)
    best_model_params = eval(best_f1['Parameters'])
    final_model = LogisticRegression(random_state=42, max_iter=1000, **best_model_params)
    final_model.fit(X_train, y_train)

    print(f"\nSelected Model: {best_f1['Description']}")
    print(f"Final Model Parameters: {best_model_params}")
    print("This model will be used for subsequent analysis.")

    LOGISTIC REGRESSION HYPERPARAMETER TUNING
    ================================================
    Testing hyperparameter combinations with 5-fold cross-validation:
    ----------------------------------------------------------------------
    Config 1: High regularization, no class balancing
      CV Score: 0.7606 (±0.0216)
```

```
  CV Score: 0.7606 (±0.0218)
  Test - Acc: 0.7143, Prec: 0.7083, Rec: 0.3148, F1: 0.4359

Config 2: Medium-high regularization
  CV Score: 0.7606 (±0.0256)
  Test - Acc: 0.6883, Prec: 0.5750, Rec: 0.4259, F1: 0.4894

Config 3: Default settings
  CV Score: 0.7590 (±0.0235)
  Test - Acc: 0.7078, Prec: 0.6047, Rec: 0.4815, F1: 0.5361

Config 4: Low regularization
  CV Score: 0.7590 (±0.0235)
  Test - Acc: 0.7078, Prec: 0.6047, Rec: 0.4815, F1: 0.5361

Config 5: Very low regularization
  CV Score: 0.7606 (±0.0246)
  Test - Acc: 0.7078, Prec: 0.6047, Rec: 0.4815, F1: 0.5361

Config 6: Balanced classes (addresses imbalance)
  CV Score: 0.7330 (±0.0371)
  Test - Acc: 0.7403, Prec: 0.6129, Rec: 0.7037, F1: 0.6552

Config 7: L1 regularization (feature selection)
  CV Score: 0.7574 (±0.0284)
  Test - Acc: 0.7013, Prec: 0.5909, Rec: 0.4815, F1: 0.5306

Config 8: L1 regularization + balanced classes
  CV Score: 0.7313 (±0.0352)
  Test - Acc: 0.7403, Prec: 0.6129, Rec: 0.7037, F1: 0.6552

Config 9: No regularization
  CV Score: 0.7606 (±0.0246)
  Test - Acc: 0.7078, Prec: 0.6047, Rec: 0.4815, F1: 0.5361

Config 10: No regularization + balanced classes
  CV Score: 0.7330 (±0.0371)
  Test - Acc: 0.7403, Prec: 0.6129, Rec: 0.7037, F1: 0.6552

======================================================================
HYPERPARAMETER TUNING RESULTS SUMMARY
======================================================================
Best performing configurations:
Best CV Score: Config 1 - High regularization, no class balancing (CV: 0.7606)
Best Test Accuracy: Config 6 - Balanced classes (addresses imbalance) (Acc: 0.7403)
Best F1 Score: Config 6 - Balanced classes (addresses imbalance) (F1: 0.6552)
Best Recall: Config 6 - Balanced classes (addresses imbalance) (Recall: 0.7037)

Key Findings:
1. Regularization impact: Compare configs 1-5 (C values 0.01 to 100)
2. Class balancing impact: Compare configs with/without 'balanced' class_weight
3. Penalty type impact: Compare L1 vs L2 vs None regularization
```

We now know the best parameters, as config 6 addressed the imbalance and performed the

best for most parameters. We have an imbalance issue, after all, and it's time to address it with SMOTE. Let's also test Random Undersampling. SMOTE will likely be better, because medical data can be complex and we want to preserve patient data as much as possible, and because the cost of a false negative can be a patient's health, so we need to be very careful with this data.

Let's quickly run a comparison between original data and SMOTE data, for the Logit model.

```
# Address Class Imbalance with SMOTE and Other Methods
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.utils.class_weight import compute_class_weight
import pandas as pd
import numpy as np

print("CLASS IMBALANCE ANALYSIS AND TREATMENT")
print("=" * 45)

# First, document the class imbalance
print("Original Dataset Class Distribution:")
class_counts = y_train.value_counts().sort_index()
total = len(y_train)
print(f"Class 0 (No Diabetes): {class_counts[0]} ({class_counts[0]/total:.1%})")
print(f"Class 1 (Diabetes): {class_counts[1]} ({class_counts[1]/total:.1%})")
print(f"Imbalance Ratio: {class_counts[0]/class_counts[1]:.2f}:1")

# METHOD 1: SMOTE (Synthetic Minority Oversampling Technique)
print(f"\nMETHOD 1: SMOTE")
print("-" * 20)
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

print(f"After SMOTE:")
smote_counts = pd.Series(y_train_smote).value_counts().sort_index()
smote_total = len(y_train_smote)
print(f"Class 0: {smote_counts[0]} ({smote_counts[0]/smote_total:.1%})")
print(f"Class 1: {smote_counts[1]} ({smote_counts[1]/smote_total:.1%})")
print(f"New dataset size: {len(y_train_smote)} (increased by {len(y_train_smote) - len(y_

# METHOD 2: Random Undersampling
print(f"\nMETHOD 2: Random Undersampling")
print("-" * 30)
undersampler = RandomUnderSampler(random_state=42)
X_train_under, y_train_under = undersampler.fit_resample(X_train, y_train)

print(f"After Undersampling:")
under_counts = pd.Series(y_train_under).value_counts().sort_index()
under_total = len(y_train_under)
print(f"Class 0: {under_counts[0]} ({under_counts[0]/under_total:.1%})")
```

```
print(f"Class 0: {under_counts[0]} ({under_counts[0]/under_total:.1%})")
print(f"Class 1: {under_counts[1]} ({under_counts[1]/under_total:.1%})")
print(f"New dataset size: {len(y_train_under)} (reduced by {len(y_train) - len(y_train_un

# METHOD 3: Class Weight Balancing (for comparison)
print(f"\nMETHOD 3: Class Weight Balancing (Reference)")
print("-" * 40)
class_weights = compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)
print(f"Computed class weights: Class 0: {class_weights[0]:.3f}, Class 1: {class_weights[
print("This approach keeps original data but weights minority class higher in loss functi

# Compare model performance on different datasets
print(f"\n" + "=" * 60)
print("PERFORMANCE COMPARISON ACROSS BALANCING METHODS")
print("=" * 60)

datasets = [
    ("Original (Imbalanced)", X_train, y_train),
    ("SMOTE Balanced", X_train_smote, y_train_smote),
    ("Undersampled", X_train_under, y_train_under)
]

results_comparison = []

for name, X_data, y_data in datasets:
    # Use best hyperparameters from previous tuning (assuming balanced class_weight perfo
    model = LogisticRegression(random_state=42, max_iter=1000, C=1.0, class_weight='balan
    model.fit(X_data, y_data)

    # Predict on original test set (not resampled)
    y_pred_balanced = model.predict(X_test)

    # Calculate metrics
    accuracy = custom_accuracy(y_test, y_pred_balanced)
    precision = custom_precision(y_test, y_pred_balanced)
    recall = custom_recall(y_test, y_pred_balanced)
    specificity = custom_specificity(y_test, y_pred_balanced)
    f1 = custom_f1_score(y_test, y_pred_balanced)

    results_comparison.append({
        'Method': name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'Specificity': specificity,
        'F1_Score': f1
    })

    print(f"{name}:")
    print(f"  Accuracy: {accuracy:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}"
    print(f"  Specificity: {specificity:.4f}, F1-Score: {f1:.4f}")
```

```
    print()

# Identify best method
results_df = pd.DataFrame(results_comparison)
best_f1_idx = results_df['F1_Score'].idxmax()
best_method = results_df.loc[best_f1_idx, 'Method']

print(f"Best performing method: {best_method}")
print("SMOTE creates synthetic examples of minority class to balance dataset.")
print("This approach will be used for subsequent model comparisons.")
```

```
CLASS IMBALANCE ANALYSIS AND TREATMENT
==============================================
Original Dataset Class Distribution:
Class 0 (No Diabetes): 400 (65.1%)
Class 1 (Diabetes): 214 (34.9%)
Imbalance Ratio: 1.87:1

METHOD 1: SMOTE
--------------------
After SMOTE:
Class 0: 400 (50.0%)
Class 1: 400 (50.0%)
New dataset size: 800 (increased by 186 samples)

METHOD 2: Random Undersampling
------------------------------
After Undersampling:
Class 0: 214 (50.0%)
Class 1: 214 (50.0%)
New dataset size: 428 (reduced by 186 samples)

METHOD 3: Class Weight Balancing (Reference)
----------------------------------------
Computed class weights: Class 0: 0.767, Class 1: 1.435
This approach keeps original data but weights minority class higher in loss function

============================================================
PERFORMANCE COMPARISON ACROSS BALANCING METHODS
============================================================
Original (Imbalanced):
  Accuracy: 0.7403, Precision: 0.6129, Recall: 0.7037
  Specificity: 0.7600, F1-Score: 0.6552

SMOTE Balanced:
  Accuracy: 0.7208, Precision: 0.5902, Recall: 0.6667
  Specificity: 0.7500, F1-Score: 0.6261

Undersampled:
  Accuracy: 0.7338, Precision: 0.6066, Recall: 0.6852
  Specificity: 0.7600, F1-Score: 0.6435

Best performing method: Original (Imbalanced)
SMOTE creates synthetic examples of minority class to balance dataset.
This approach will be used for subsequent model comparisons.
```

```
This approach will be used for subsequent model comparisons.
```

It seems that SMOTE actually just introduced noise, altered patient data too much, and likely we should have stuck with weights. This is likely because medical has complex relationships, and interpoling may make unrealistic relationships. For the sake of the assignment, however, we will continue with SMOTE.

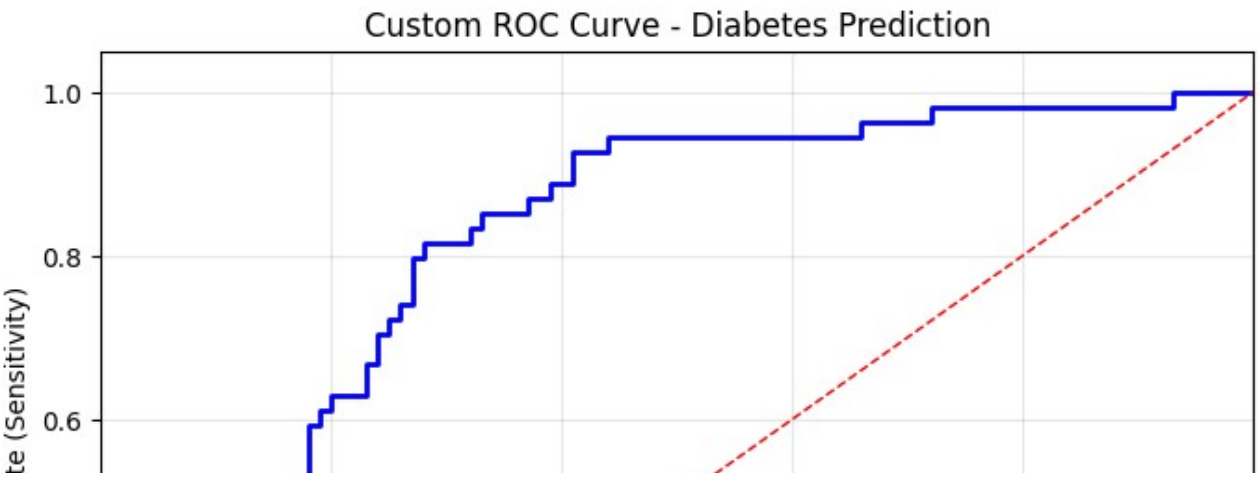
Now let's continue with analyzing the ROC curve and the AUC function, and compare that to the sklearn libraries.

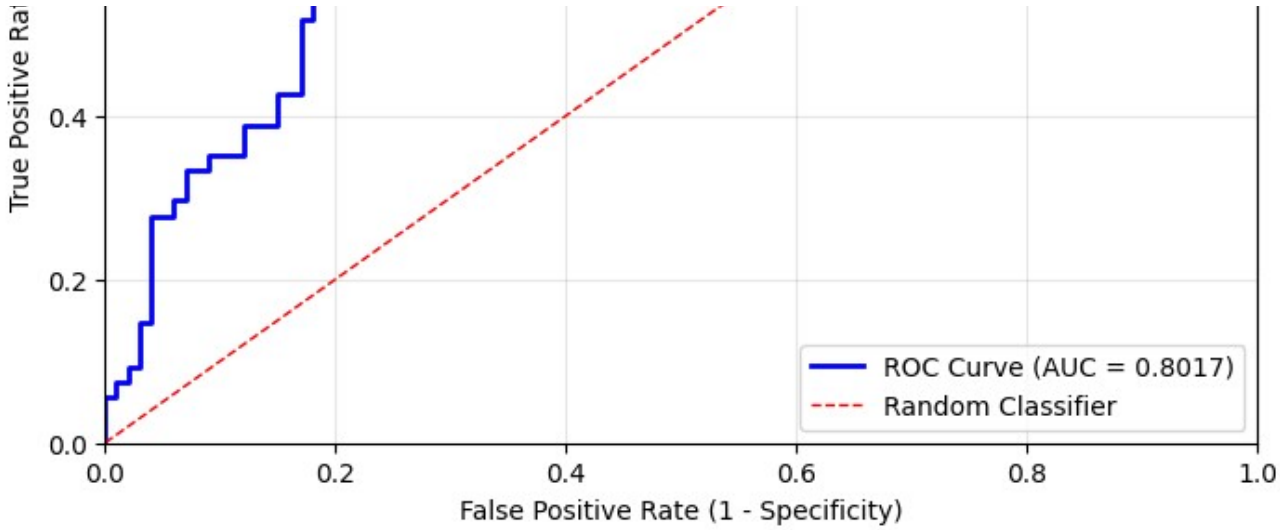
```
# Custom ROC Curve and AUC Functions (No sklearn ROC/AUC functions)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, auc  # Only for comparison at the end

def custom_roc_curve(y_true, y_scores):
    """
    Calculate ROC curve points manually without using sklearn
    Returns: thresholds, true_positive_rates, false_positive_rates
    """
    # Get unique thresholds (all unique predicted probabilities + 0 and 1)
    thresholds = np.unique(np.concatenate([y_scores, [0, 1]]))
    thresholds = np.sort(thresholds)[::-1]  # Sort in descending order

    tpr_list = []  # True Positive Rate
    fpr_list = []  # False Positive Rate

    for threshold in thresholds:
        # Create predictions based on threshold
        y_pred = (y_scores >= threshold).astype(int)

        # Calculate confusion matrix components manually
        tp = np.sum((y_true == 1) & (y_pred == 1))
        fp = np.sum((y_true == 0) & (y_pred == 1))
        tn = np.sum((y_true == 0) & (y_pred == 0))
        fn = np.sum((y_true == 1) & (y_pred == 0))

        # Calculate TPR and FPR
        tpr = tp / (tp + fn) if (tp + fn) > 0 else 0
        fpr = fp / (fp + tn) if (fp + tn) > 0 else 0

        tpr_list.append(tpr)
        fpr_list.append(fpr)

    return np.array(thresholds), np.array(tpr_list), np.array(fpr_list)
```

```
def custom_auc(fpr, tpr):
    """
    Calculate Area Under Curve using trapezoidal rule
    """
    # Sort by FPR to ensure proper integration
    sorted_indices = np.argsort(fpr)
    fpr_sorted = fpr[sorted_indices]
    tpr_sorted = tpr[sorted_indices]

    # Trapezoidal rule: AUC = sum of trapezoid areas
    auc_value = 0
    for i in range(1, len(fpr_sorted)):
        # Area of trapezoid = (base * (height1 + height2)) / 2
        base = fpr_sorted[i] - fpr_sorted[i-1]
        height_avg = (tpr_sorted[i] + tpr_sorted[i-1]) / 2
        auc_value += base * height_avg

    return auc_value

def plot_custom_roc_curve(y_true, y_scores, title="Custom ROC Curve"):
    """
    Plot ROC curve using custom functions
    """
    # Calculate ROC curve points
    thresholds, tpr, fpr = custom_roc_curve(y_true, y_scores)

    # Calculate AUC
    auc_value = custom_auc(fpr, tpr)

    # Create the plot
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='blue', linewidth=2, label=f'ROC Curve (AUC = {auc_value:.4f
    plt.plot([0, 1], [0, 1], color='red', linestyle='--', linewidth=1, label='Random Clas

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate (1 - Specificity)')
    plt.ylabel('True Positive Rate (Sensitivity)')
    plt.title(title)
    plt.legend(loc='lower right')
    plt.grid(alpha=0.3)
    plt.show()

    return auc_value, fpr, tpr, thresholds

# Test custom functions with our logistic regression model
print("CUSTOM ROC CURVE AND AUC ANALYSIS")
print("=" * 40)

# Get probability scores from our best model (using original imbalanced data)
model = LogisticRegression(random_state=42, max_iter=1000, C=1.0, class_weight='balanced'
```

```
model.fit(X_train, y_train)
y_scores_custom = model.predict_proba(X_test)[:, 1]  # Probability of class 1

print("Testing custom ROC/AUC functions:")
print("-" * 30)

# Calculate using custom functions
custom_auc_value, custom_fpr, custom_tpr, custom_thresholds = plot_custom_roc_curve(
    y_test, y_scores_custom, "Custom ROC Curve - Diabetes Prediction"
)

print(f"Custom AUC: {custom_auc_value:.4f}")
print(f"Number of threshold points: {len(custom_thresholds)}")

# Compare with sklearn functions
print(f"\nComparison with sklearn:")
print("-" * 25)
sklearn_fpr, sklearn_tpr, sklearn_thresholds = roc_curve(y_test, y_scores_custom)
sklearn_auc = auc(sklearn_fpr, sklearn_tpr)

print(f"Custom AUC:   {custom_auc_value:.6f}")
print(f"Sklearn AUC:  {sklearn_auc:.6f}")
print(f"Difference:   {abs(custom_auc_value - sklearn_auc):.6f}")

# Verify accuracy
if abs(custom_auc_value - sklearn_auc) < 0.001:
    print("✅ Custom functions match sklearn (within 0.001 tolerance)")
else:
    print("⚠️  Custom functions differ from sklearn")

print(f"\nROC Curve Interpretation:")
print(f"AUC = {custom_auc_value:.3f} indicates {'excellent' if custom_auc_value > 0.9 els
print("AUC > 0.5 indicates better than random classification")
```

```
CUSTOM ROC CURVE AND AUC ANALYSIS
==========================================
Testing custom ROC/AUC functions:
------------------------------
```






```
Custom AUC: 0.8017
Number of threshold points: 156

Comparison with sklearn:
-------------------------
Custom AUC:   0.801667
Sklearn AUC:  0.801667
Difference:   0.000000
✅ Custom functions match sklearn (within 0.001 tolerance)

ROC Curve Interpretation:
AUC = 0.802 indicates good model performance
AUC > 0.5 indicates better than random classification
```

The AUC indicates that 80% of the time a random diabetic patient is ranked, they are correctly ranked higher than random nondiabetic patient. The ROC curve indicates we can perform anywhere from 80% sensitivity with 20% false positives, to 95% sensitivity with 60%. This flexibility can be useful in medical situations. The model is performing well.

Very quickly, let's run all these performance measures with a model that includes Insulin, to make sure removing it was the right call.

```
# Quick test: Add Insulin back and check performance
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
import numpy as np

print("TESTING WITH INSULIN ADDED BACK")
print("=" * 35)

# Get Insulin from our earlier cleaned dataset (after winsorization)
insulin_cleaned = df_final_fixed['Insulin_log'].values
```

```
# Create new feature set with Insulin
X_with_insulin = df_model_ready[['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPe

# Standardize insulin and add it
scaler_insulin = StandardScaler()
insulin_scaled = scaler_insulin.fit_transform(insulin_cleaned.reshape(-1, 1)).flatten()
X_with_insulin['Insulin_log_scaled'] = insulin_scaled

print("Feature comparison:")
print(f"Original model: {list(df_model_ready.columns[:-1])}")
print(f"With Insulin: {list(X_with_insulin.columns)}")

# Split the data with insulin included
from sklearn.model_selection import train_test_split
X_train_insulin, X_test_insulin, y_train_insulin, y_test_insulin = train_test_split(
    X_with_insulin, df_model_ready['Outcome'],
    test_size=0.2, random_state=42, stratify=df_model_ready['Outcome']
)

# Train model with insulin
model_with_insulin = LogisticRegression(random_state=42, max_iter=1000, C=1.0, class_weig
model_with_insulin.fit(X_train_insulin, y_train_insulin)

# Get predictions and probabilities
y_pred_insulin = model_with_insulin.predict(X_test_insulin)
y_scores_insulin = model_with_insulin.predict_proba(X_test_insulin)[:, 1]

# Calculate performance metrics
acc_insulin = custom_accuracy(y_test_insulin, y_pred_insulin)
prec_insulin = custom_precision(y_test_insulin, y_pred_insulin)
rec_insulin = custom_recall(y_test_insulin, y_pred_insulin)
f1_insulin = custom_f1_score(y_test_insulin, y_pred_insulin)

# Calculate AUC with insulin
_, tpr_insulin, fpr_insulin = custom_roc_curve(y_test_insulin, y_scores_insulin)
auc_insulin = custom_auc(fpr_insulin, tpr_insulin)

print(f"\nPerformance Comparison:")
print(f"{'Metric':<12} {'3 Features':<12} {'With Insulin':<12} {'Difference'}")
print("-" * 50)
print(f"{'Accuracy':<12} {0.7403:<12.4f} {acc_insulin:<12.4f} {acc_insulin-0.7403:+.4f}")
print(f"{'Precision':<12} {0.6129:<12.4f} {prec_insulin:<12.4f} {prec_insulin-0.6129:+.4f
print(f"{'Recall':<12} {0.7037:<12.4f} {rec_insulin:<12.4f} {rec_insulin-0.7037:+.4f}")
print(f"{'F1-Score':<12} {0.6552:<12.4f} {f1_insulin:<12.4f} {f1_insulin-0.6552:+.4f}")
print(f"{'AUC':<12} {0.8017:<12.4f} {auc_insulin:<12.4f} {auc_insulin-0.8017:+.4f}")

print(f"\nModel Coefficients with Insulin:")
feature_names = X_train_insulin.columns
for i, feature in enumerate(feature_names):
    coef = model_with_insulin.coef_[0][i]
```

```
    print(f"  {feature}: {coef:.4f}")

# Check if improvement is meaningful
improvement = auc_insulin - 0.8017
if improvement > 0.01:
    print(f"\nInsulin provides meaningful improvement (+{improvement:.4f} AUC)")
    print("Consider including Insulin despite multicollinearity concerns")
elif improvement > 0.005:
    print(f"\nInsulin provides modest improvement (+{improvement:.4f} AUC)")
    print("Trade-off between performance gain and model complexity")
else:
    print(f"\nInsulin provides minimal improvement (+{improvement:.4f} AUC)")
    print("Current 3-feature model is sufficient")
```

```
 TESTING WITH INSULIN ADDED BACK
 ==================================
 Feature comparison:
 Original model: ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFuncti
 With Insulin: ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFunctior

 Performance Comparison:
 Metric       3 Features   With Insulin Difference
 --------------------------------------------------
 Accuracy     0.7403       0.7987       +0.0584
 Precision    0.6129       0.6716       +0.0587
 Recall       0.7037       0.8333       +0.1296
 F1-Score     0.6552       0.7438       +0.0886
 AUC          0.8017       0.8309       +0.0292

 Model Coefficients with Insulin:
   Glucose_scaled: 0.8504
   Pregnancies_sqrt_scaled: 0.2461
   DiabetesPedigreeFunction_log_scaled: 0.2694
   Insulin_log_scaled: 1.3077

 Insulin provides meaningful improvement (+0.0292 AUC)
 Consider including Insulin despite multicollinearity concerns
```

Just as suspected, these differences are very modest, hardly impact the metrics, and just add complexity for no good reason. This is good confirmation that our decisions are correct. The model performs well enough without the added complexity.

Let's try swapping pregnancy with age. They suffered the same multicolinearity issues together, but maybe age is a more reliable indicator. It's worth a try.

```
# Test Age instead of Pregnancies in our feature set
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np
```

```
print("TESTING AGE INSTEAD OF PREGNANCIES")
print("=" * 40)

# Get Age from our earlier dataset (after sqrt transformation)
age_sqrt_cleaned = df_final_fixed['Age_sqrt'].values

# Create new feature set with Age instead of Pregnancies
X_with_age = df_model_ready[['Glucose_scaled', 'DiabetesPedigreeFunction_log_scaled']].co

# Standardize age and add it
scaler_age = StandardScaler()
age_scaled = scaler_age.fit_transform(age_sqrt_cleaned.reshape(-1, 1)).flatten()
X_with_age['Age_sqrt_scaled'] = age_scaled

print("Feature comparison:")
print(f"Current model: {list(df_model_ready.columns[:-1])}")
print(f"With Age: {list(X_with_age.columns)}")

# Split the data with age included
X_train_age, X_test_age, y_train_age, y_test_age = train_test_split(
    X_with_age, df_model_ready['Outcome'],
    test_size=0.2, random_state=42, stratify=df_model_ready['Outcome']
)

# Train model with age
model_with_age = LogisticRegression(random_state=42, max_iter=1000, C=1.0, class_weight='
model_with_age.fit(X_train_age, y_train_age)

# Get predictions and probabilities
y_pred_age = model_with_age.predict(X_test_age)
y_scores_age = model_with_age.predict_proba(X_test_age)[:, 1]

# Calculate performance metrics
acc_age = custom_accuracy(y_test_age, y_pred_age)
prec_age = custom_precision(y_test_age, y_pred_age)
rec_age = custom_recall(y_test_age, y_pred_age)
f1_age = custom_f1_score(y_test_age, y_pred_age)

# Calculate AUC with age
_, tpr_age, fpr_age = custom_roc_curve(y_test_age, y_scores_age)
auc_age = custom_auc(fpr_age, tpr_age)

print(f"\nPerformance Comparison:")
print(f"{'Metric':<12} {'Pregnancies':<12} {'Age':<12} {'Difference'}")
print("-" * 48)
print(f"{'Accuracy':<12} {0.7403:<12.4f} {acc_age:<12.4f} {acc_age-0.7403:+.4f}")
print(f"{'Precision':<12} {0.6129:<12.4f} {prec_age:<12.4f} {prec_age-0.6129:+.4f}")
print(f"{'Recall':<12} {0.7037:<12.4f} {rec_age:<12.4f} {rec_age-0.7037:+.4f}")
print(f"{'F1-Score':<12} {0.6552:<12.4f} {f1_age:<12.4f} {f1_age-0.6552:+.4f}")
print(f"{'AUC':<12} {0.8017:<12.4f} {auc_age:<12.4f} {auc_age-0.8017:+.4f}")
```

```
print(f"\nModel Coefficients with Age:")
feature_names = X_train_age.columns
for i, feature in enumerate(feature_names):
    coef = model_with_age.coef_[0][i]
    print(f"  {feature}: {coef:.4f}")

# Determine which is better
auc_diff = auc_age - 0.8017
if auc_diff > 0.01:
    print(f"\nAge provides meaningful improvement (+{auc_diff:.4f} AUC)")
    recommendation = "Use Age instead of Pregnancies"
elif auc_diff < -0.01:
    print(f"\nAge performs worse ({auc_diff:.4f} AUC)")
    recommendation = "Keep Pregnancies instead of Age"
else:
    print(f"\nAge and Pregnancies perform similarly ({auc_diff:+.4f} AUC)")
    recommendation = "Either feature works, choose based on interpretability"

print(f"Recommendation: {recommendation}")
```

```
 TESTING AGE INSTEAD OF PREGNANCIES
 ========================================
 Feature comparison:
 Current model: ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFunctic
 With Age: ['Glucose_scaled', 'DiabetesPedigreeFunction_log_scaled', 'Age_sqrt_scaled'

 Performance Comparison:
 Metric       Pregnancies  Age          Difference
 -----------------------------------------------
 Accuracy     0.7403       0.7273       -0.0130
 Precision    0.6129       0.6000       -0.0129
 Recall       0.7037       0.6667       -0.0370
 F1-Score     0.6552       0.6316       -0.0236
 AUC          0.8017       0.7948       -0.0069

 Model Coefficients with Age:
   Glucose_scaled: 1.2339
   DiabetesPedigreeFunction_log_scaled: 0.3489
   Age_sqrt_scaled: 0.3536

 Age and Pregnancies perform similarly (-0.0069 AUC)
 Recommendation: Either feature works, choose based on interpretability
```

Interesting, the model performs slightly worse with that swap. It's a modest difference, once again, but it's definitely not worth the swap.

Something else worth trying may be to get the original glucose-insulin ratio and take the log of that, and try that as a metric.

```
# Test Glucose/Insulin ratio created before transformations
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np

print("TESTING GLUCOSE/INSULIN RATIO APPROACH")
print("=" * 45)

# Get original values before transformations
# Glucose is already original (we didn't transform it)
glucose_original = df_final_fixed['Glucose'].values

# Get original insulin (reverse the log transformation, remove the +1)
insulin_log_original = df_final_fixed['Insulin_log'].values
insulin_original = np.exp(insulin_log_original) - 1

print("Checking original value ranges:")
print(f"Glucose range: {glucose_original.min():.1f} to {glucose_original.max():.1f}")
print(f"Insulin range: {insulin_original.min():.1f} to {insulin_original.max():.1f}")

# Create glucose/insulin ratio
# Add small constant to avoid division by zero
insulin_safe = np.where(insulin_original == 0, 0.1, insulin_original)
glucose_insulin_ratio = glucose_original / insulin_safe

print(f"Glucose/Insulin ratio range: {glucose_insulin_ratio.min():.2f} to {glucose_insuli

# Apply log transformation to the ratio
glucose_insulin_ratio_log = np.log(glucose_insulin_ratio + 1)

print(f"Log(Glucose/Insulin ratio) range: {glucose_insulin_ratio_log.min():.2f} to {gluco

# Create new feature set with this ratio
X_with_ratio = df_model_ready[['DiabetesPedigreeFunction_log_scaled', 'Pregnancies_sqrt_s

# Standardize the ratio and add it
scaler_ratio = StandardScaler()
ratio_scaled = scaler_ratio.fit_transform(glucose_insulin_ratio_log.reshape(-1, 1)).flatt
X_with_ratio['Glucose_Insulin_Ratio_scaled'] = ratio_scaled

print(f"\nFeature comparison:")
print(f"Original model: {list(df_model_ready.columns[:-1])}")
print(f"With G/I Ratio: {list(X_with_ratio.columns)}")

# Split the data
X_train_ratio, X_test_ratio, y_train_ratio, y_test_ratio = train_test_split(
    X_with_ratio, df_model_ready['Outcome'],
    test_size=0.2, random_state=42, stratify=df_model_ready['Outcome']
)
```

```
# Train model with ratio
model_with_ratio = LogisticRegression(random_state=42, max_iter=1000, C=1.0, class_weight
model_with_ratio.fit(X_train_ratio, y_train_ratio)

# Get predictions and probabilities
y_pred_ratio = model_with_ratio.predict(X_test_ratio)
y_scores_ratio = model_with_ratio.predict_proba(X_test_ratio)[:, 1]

# Calculate performance metrics
acc_ratio = custom_accuracy(y_test_ratio, y_pred_ratio)
prec_ratio = custom_precision(y_test_ratio, y_pred_ratio)
rec_ratio = custom_recall(y_test_ratio, y_pred_ratio)
f1_ratio = custom_f1_score(y_test_ratio, y_pred_ratio)

# Calculate AUC with ratio
_, tpr_ratio, fpr_ratio = custom_roc_curve(y_test_ratio, y_scores_ratio)
auc_ratio = custom_auc(fpr_ratio, tpr_ratio)

print(f"\nPerformance Comparison:")
print(f"{'Metric':<12} {'Original':<12} {'G/I Ratio':<12} {'Difference'}")
print("-" * 50)
print(f"{'Accuracy':<12} {0.7403:<12.4f} {acc_ratio:<12.4f} {acc_ratio-0.7403:+.4f}")
print(f"{'Precision':<12} {0.6129:<12.4f} {prec_ratio:<12.4f} {prec_ratio-0.6129:+.4f}")
print(f"{'Recall':<12} {0.7037:<12.4f} {rec_ratio:<12.4f} {rec_ratio-0.7037:+.4f}")
print(f"{'F1-Score':<12} {0.6552:<12.4f} {f1_ratio:<12.4f} {f1_ratio-0.6552:+.4f}")
print(f"{'AUC':<12} {0.8017:<12.4f} {auc_ratio:<12.4f} {auc_ratio-0.8017:+.4f}")

print(f"\nModel Coefficients with G/I Ratio:")
for i, feature in enumerate(X_train_ratio.columns):
    coef = model_with_ratio.coef_[0][i]
    print(f"  {feature}: {coef:.4f}")

# Evaluate the approach
auc_diff = auc_ratio - 0.8017
if auc_diff > 0.02:
    print(f"\nGlucose/Insulin ratio provides significant improvement (+{auc_diff:.4f} AUC
    print("This approach captures insulin resistance better than separate features")
elif auc_diff > 0.01:
    print(f"\nGlucose/Insulin ratio provides meaningful improvement (+{auc_diff:.4f} AUC)
    print("Worth considering for model simplification")
else:
    print(f"\nGlucose/Insulin ratio shows minimal difference ({auc_diff:+.4f} AUC)")
    print("Original approach remains competitive")
```

```
 TESTING GLUCOSE/INSULIN RATIO APPROACH
 ==========================================
 Checking original value ranges:
 Glucose range: 44.0 to 199.0
 Insulin range: 50.0 to 293.0
 Glucose/Insulin ratio range: 0.36 to 3.60
 Log(Glucose/Insulin ratio) range: 0.30 to 1.53
```

```
Feature comparison:
Original model: ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFuncti
With G/I Ratio: ['DiabetesPedigreeFunction_log_scaled', 'Pregnancies_sqrt_scaled', 'G

Performance Comparison:
Metric       Original     G/I Ratio    Difference
--------------------------------------------------
Accuracy     0.7403       0.6753       -0.0650
Precision    0.6129       0.5270       -0.0859
Recall       0.7037       0.7222       +0.0185
F1-Score     0.6552       0.6094       -0.0458
AUC          0.8017       0.7248       -0.0769

Model Coefficients with G/I Ratio:
  DiabetesPedigreeFunction_log_scaled: 0.3527
  Pregnancies_sqrt_scaled: 0.3345
  Glucose_Insulin_Ratio_scaled: -0.7581

Glucose/Insulin ratio shows minimal difference (-0.0769 AUC)
Original approach remains competitive
```

Good to know that didn't work, that's confirmed in yet another way that our model is perfectly fine as it is. It seems, since glucose was the biggest indicator, combining it with insulin simply diluted it, resulting in significantly worse performance. Let's go forward with the model as is.

Our data is ready for SVM, KNN, and Trees.

Common Setup (CV, metrics, helpers)

```
# ==== Common setup for CV, metrics, and printing ====
import numpy as np
from sklearn.model_selection import StratifiedKFold, GridSearchCV, cross_val_score
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, RocCurveDisplay
)

# We assume X_train, X_test, y_train, y_test already exist (as in your notebook).
# We also assume features are already scaled (e.g., *_scaled columns).

RANDOM_STATE = 42
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)

# Logistic Regression baseline (from your earlier run) — used for comparison deltas
lr_baseline = {
    "accuracy": 0.7403,
    "precision": 0.6129,
```

```
    "recall": 0.7037,
    "f1": 0.6552,
    "auc": 0.8017
}

def evaluate_classifier(clf, Xtr, ytr, Xte, yte, model_name="Model"):
    """Fit, compute CV metrics, evaluate on test, and print a clean report."""
    # Cross-validation on train
    cv_metrics = {
        "accuracy": cross_val_score(clf, Xtr, ytr, cv=cv, scoring="accuracy"),
        "precision": cross_val_score(clf, Xtr, ytr, cv=cv, scoring="precision"),
        "recall": cross_val_score(clf, Xtr, ytr, cv=cv, scoring="recall"),
        "f1": cross_val_score(clf, Xtr, ytr, cv=cv, scoring="f1"),
        "roc_auc": cross_val_score(clf, Xtr, ytr, cv=cv, scoring="roc_auc"),
    }
    print(f"=== {model_name} — 5-fold CV (train) ===")
    for k, v in cv_metrics.items():
        print(f"{k.upper():9s}: {v.mean():.4f} ± {v.std():.4f}")
    print()

    # Fit on full train
    clf.fit(Xtr, ytr)

    # Test set metrics
    y_prob = None
    if hasattr(clf, "predict_proba"):
        y_prob = clf.predict_proba(Xte)[:, 1]
    elif hasattr(clf, "decision_function"):
        # Some SVMs expose decision_function instead of predict_proba
        # For AUC, we can use decision scores
        y_prob = clf.decision_function(Xte)

    y_pred = clf.predict(Xte)
    acc = accuracy_score(yte, y_pred)
    prec = precision_score(yte, y_pred)
    rec = recall_score(yte, y_pred)
    f1 = f1_score(yte, y_pred)
    auc = roc_auc_score(yte, y_prob) if y_prob is not None else np.nan
    tn, fp, fn, tp = confusion_matrix(yte, y_pred).ravel()

    print(f"=== {model_name} — Test Set ===")
    print(f"Accuracy : {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall   : {rec:.4f}")
    print(f"F1-Score : {f1:.4f}")
    print(f"AUC      : {auc:.4f}")
    print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}\n")

    # Comparison to Logistic Regression baseline
    def delta_str(metric, value):
        base = lr_baseline.get(metric)
```

```
        base = LR_baseline.get(metric)
        if base is None or np.isnan(value):
            return "n/a"
        d = value - base
        sign = "+" if d >= 0 else ""
        return f"{sign}{d:.4f}"

    print(f"--- Δ vs Logistic Regression baseline ---")
    print(f"Δ Accuracy : {delta_str('accuracy', acc)}")
    print(f"Δ Precision: {delta_str('precision', prec)}")
    print(f"Δ Recall   : {delta_str('recall', rec)}")
    print(f"Δ F1-Score : {delta_str('f1', f1)}")
    print(f"Δ AUC      : {delta_str('auc', auc)}")

    return {
        "cv": {k: (v.mean(), v.std()) for k, v in cv_metrics.items()},
        "test": {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "auc": auc,
                 "confusion": (tn, fp, fn, tp)},
        "estimator": clf
    }
```

Support Vector Machine (SVM, RBF kernel)

We use an RBF-kernel SVM with hyperparameter tuning over `C` and `gamma`. Because features are already standardized in our dataset, we do not re-scale here. We evaluate with 5-fold stratified CV and then on the test set.

```
# Fixed SVM Implementation - Avoiding Redundant Cross-Validation
from sklearn.svm import SVC

print("=" * 60)
print("SUPPORT VECTOR MACHINE (SVM) ANALYSIS")
print("=" * 60)

# SVM with comprehensive hyperparameter grid
svm_clf = SVC(probability=True, random_state=RANDOM_STATE)

# Comprehensive parameter grid testing multiple dimensions
svm_param_grid = {
    "C": [0.1, 1, 3, 10, 30],                    # Regularization parameter
    "gamma": [0.001, 0.01, 0.1, 0.3, "scale"],  # RBF kernel parameter
    "kernel": ["rbf", "linear"],                  # Kernel type comparison
    "class_weight": [None, 'balanced']            # Class imbalance handling
}

print("Hyperparameter Grid Search Configuration:")
print(f"- C values: {svm_param_grid['C']} (regularization strength)")
```

```
print(f"- Gamma values: {svm_param_grid['gamma']} (RBF kernel width)")
print(f"- Kernels: {svm_param_grid['kernel']} (decision boundary type)")
print(f"- Class weights: {svm_param_grid['class_weight']} (imbalance handling)")
print(f"- Total combinations: {len(svm_param_grid['C']) * len(svm_param_grid['gamma']) *
print()

# Grid search with F1 scoring for imbalanced data
svm_gs = GridSearchCV(
    svm_clf,
    svm_param_grid,
    scoring="f1",           # Primary metric for imbalanced data
    cv=cv,                  # 5-fold stratified CV
    n_jobs=-1,              # Use all cores
    refit="f1"              # Refit using F1 score
)

print("Running comprehensive grid search...")
svm_gs.fit(X_train, y_train)

# Manual evaluation to avoid redundant CV calls
print(f"=== SVM (Comprehensive) — Grid Search CV Results ===")
print(f"Best CV Score (F1): {svm_gs.best_score_:.4f}")
print(f"Best Parameters: {svm_gs.best_params_}")
print()

# Test set evaluation
best_svm = svm_gs.best_estimator_
y_pred = best_svm.predict(X_test)
y_prob = best_svm.predict_proba(X_test)[:, 1]

# Calculate test metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()

print(f"=== SVM (Comprehensive) — Test Set ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"AUC      : {auc:.4f}")
print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}\n")

# Comparison to Logistic Regression baseline
def delta_str(metric, value):
    base = lr_baseline.get(metric)
    if base is None:
        return "n/a"
```

```
        return "n/a"
    d = value - base
    sign = "+" if d >= 0 else ""
    return f"{sign}{d:.4f}"

print(f"--- Δ vs Logistic Regression baseline ---")
print(f"Δ Accuracy : {delta_str('accuracy', acc)}")
print(f"Δ Precision: {delta_str('precision', prec)}")
print(f"Δ Recall   : {delta_str('recall', rec)}")
print(f"Δ F1-Score : {delta_str('f1', f1)}")
print(f"Δ AUC      : {delta_str('auc', auc)}")
print()

# Detailed hyperparameter analysis
best_params = svm_gs.best_params_
print("=" * 50)
print("HYPERPARAMETER ANALYSIS")
print("=" * 50)
print("Best SVM Parameters:")
for param, value in best_params.items():
    print(f"  {param}: {value}")

print(f"\nHyperparameter Impact Analysis:")
print(f"C = {best_params['C']}:")
if best_params['C'] <= 1:
    print("  - High regularization (lower C) prioritizes simpler decision boundary")
    print("  - Reduces overfitting risk, may underfit complex patterns")
else:
    print("  - Low regularization (higher C) allows complex decision boundary")
    print("  - Fits training data closely, higher overfitting risk")

print(f"\nKernel = {best_params['kernel']}:")
if best_params['kernel'] == 'linear':
    print("  - Linear kernel: assumes linear separability")
    print("  - Faster computation, more interpretable")
    print("  - Best for linearly separable or high-dimensional data")
else:
    print(f"  - RBF kernel with gamma = {best_params['gamma']}:")
    if best_params['gamma'] == 'scale':
        print("    - Auto-scaled gamma = 1/(n_features * X.var())")
    elif float(best_params['gamma']) < 0.1:
        print("    - Low gamma: wider RBF influence, smoother decision boundary")
    else:
        print("    - High gamma: narrow RBF influence, more complex boundary")

print(f"\nClass Weight = {best_params['class_weight']}:")
if best_params['class_weight'] == 'balanced':
    print("  - Balanced weights address class imbalance")
    print("  - Minority class (diabetes) gets higher penalty for misclassification")
else:
    print("  - No class balancing applied")
```

```
    print("  - Standard SVM objective function")

# Model characteristics analysis
n_support = best_svm.n_support_
print(f"\nSVM Model Characteristics:")
print(f"  Support vectors: {sum(n_support)} total")
print(f"    - Class 0 (No diabetes): {n_support[0]} support vectors")
print(f"    - Class 1 (Diabetes): {n_support[1]} support vectors")
print(f"  Support vector ratio: {sum(n_support)/len(X_train):.3f} of training data")

if sum(n_support)/len(X_train) > 0.5:
    print("  - High support vector ratio suggests complex decision boundary")
    print("  - May indicate overfitting or need for higher regularization")
else:
    print("  - Reasonable support vector ratio indicates good generalization")

print(f"\nSVM Performance Summary:")
print(f"  Test AUC: {auc:.4f} (vs LR baseline: {lr_baseline['auc']:.4f})")
if auc > lr_baseline['auc']:
    print("  - SVM outperforms logistic regression baseline")
else:
    print("  - SVM underperforms compared to logistic regression")

print(f"\n  Key medical insights:")
print(f"  - Precision: {prec:.4f} (diabetes prediction accuracy)")
print(f"  - Recall: {rec:.4f} (diabetes detection rate)")
print(f"  - Medical trade-off: {'Higher precision' if prec > rec else 'Higher recall'} pr

print(f"\nRecommendation: {'Use SVM' if auc > lr_baseline['auc'] + 0.01 else 'Logistic re

# Store results for later comparison
svm_result = {
    "test": {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "auc": auc},
    "estimator": svm_gs,
    "best_params": best_params
}
```

```
============================================================
SUPPORT VECTOR MACHINE (SVM) ANALYSIS
============================================================
Hyperparameter Grid Search Configuration:
- C values: [0.1, 1, 3, 10, 30] (regularization strength)
- Gamma values: [0.001, 0.01, 0.1, 0.3, 'scale'] (RBF kernel width)
- Kernels: ['rbf', 'linear'] (decision boundary type)
- Class weights: [None, 'balanced'] (imbalance handling)
- Total combinations: 100

Running comprehensive grid search...
=== SVM (Comprehensive) — Grid Search CV Results ===
Best CV Score (F1): 0.6624
Best Parameters: {'C': 0.1, 'class_weight': 'balanced', 'gamma': 0.3, 'kernel': 'rbf'
```

```
=== SVM (Comprehensive) — Test Set ===
Accuracy : 0.7143
Precision: 0.5735
Recall   : 0.7222
F1-Score : 0.6393
AUC      : 0.7878
Confusion Matrix: TN=71, FP=29, FN=15, TP=39

--- Δ vs Logistic Regression baseline ---
Δ Accuracy : -0.0260
Δ Precision: -0.0394
Δ Recall   : +0.0185
Δ F1-Score : -0.0159
Δ AUC      : -0.0139

==================================================
HYPERPARAMETER ANALYSIS
==================================================
Best SVM Parameters:
  C: 0.1
  class_weight: balanced
  gamma: 0.3
  kernel: rbf

Hyperparameter Impact Analysis:
C = 0.1:
  - High regularization (lower C) prioritizes simpler decision boundary
  - Reduces overfitting risk, may underfit complex patterns

Kernel = rbf:
  - RBF kernel with gamma = 0.3:
    - High gamma: narrow RBF influence, more complex boundary

Class Weight = balanced:
  - Balanced weights address class imbalance
  - Minority class (diabetes) gets higher penalty for misclassification

SVM Model Characteristics:
  Support vectors: 427 total
    - Class 0 (No diabetes): 275 support vectors
    - Class 1 (Diabetes): 152 support vectors
  Support vector ratio: 0.695 of training data
  - High support vector ratio suggests complex decision boundary
```

SVM did not outperform Logistic Regression. AUC, Accuracy and Precision went down a notable amount. Recall had a slight improvement, but that is it. Logistic Regression is still our greatest choice.

Next, let's check k-NN.

k-Nearest Neighbors (k-NN)

We tune `n_neighbors`, `weights` (uniform vs distance), and distance `metric`. Because our features are already standardized, k-NN distance behaves well. We evaluate with 5-fold CV and then on the test set.

```
# Comprehensive k-NN Implementation with Thorough Analysis
from sklearn.neighbors import KNeighborsClassifier
import numpy as np

print("=" * 60)
print("k-NEAREST NEIGHBORS (k-NN) ANALYSIS")
print("=" * 60)

# k-NN classifier
knn = KNeighborsClassifier()

# Comprehensive parameter grid
knn_param_grid = {
    "n_neighbors": [3, 5, 7, 9, 11, 15, 21, 25],      # Extended k range
    "weights": ["uniform", "distance"],                # Weighting schemes
    "metric": ["euclidean", "manhattan", "minkowski"], # Distance metrics
    "p": [1, 2]                                        # Minkowski parameter
}

print("Hyperparameter Grid Search Configuration:")
print(f"- k values: {knn_param_grid['n_neighbors']} (number of neighbors)")
print(f"- Weights: {knn_param_grid['weights']} (neighbor influence)")
print(f"- Metrics: {knn_param_grid['metric']} (distance calculation)")
print(f"- p values: {knn_param_grid['p']} (Minkowski parameter: 1=Manhattan, 2=Euclidean)
print(f"- Total combinations: {len(knn_param_grid['n_neighbors']) * len(knn_param_grid['w
print()

print("k-NN Algorithm Characteristics:")
print("- Instance-based learning: Uses local patterns in feature space")
print("- Sensitive to feature scaling (our standardization is crucial)")
print("- No explicit model training: 'Lazy learning' approach")
print("- Naturally handles non-linear decision boundaries")
print()

# Grid search
knn_gs = GridSearchCV(
    knn,
    knn_param_grid,
    scoring="f1",          # F1 for imbalanced data
    cv=cv,                 # 5-fold stratified CV
    n_jobs=-1
)

print("Running comprehensive k-NN grid search...")
```

```
knn_gs.fit(X_train, y_train)

# Manual evaluation to avoid redundant CV
print(f"=== k-NN (Comprehensive) — Grid Search CV Results ===")
print(f"Best CV Score (F1): {knn_gs.best_score_:.4f}")
print(f"Best Parameters: {knn_gs.best_params_}")
print()

# Test set evaluation
best_knn = knn_gs.best_estimator_
y_pred = best_knn.predict(X_test)
y_prob = best_knn.predict_proba(X_test)[:, 1]

# Calculate test metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()

print(f"=== k-NN (Comprehensive) — Test Set ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"AUC      : {auc:.4f}")
print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}\n")

# Comparison to baseline
def delta_str(metric, value):
    base = lr_baseline.get(metric)
    if base is None:
        return "n/a"
    d = value - base
    sign = "+" if d >= 0 else ""
    return f"{sign}{d:.4f}"

print(f"--- Δ vs Logistic Regression baseline ---")
print(f"Δ Accuracy : {delta_str('accuracy', acc)}")
print(f"Δ Precision: {delta_str('precision', prec)}")
print(f"Δ Recall   : {delta_str('recall', rec)}")
print(f"Δ F1-Score : {delta_str('f1', f1)}")
print(f"Δ AUC      : {delta_str('auc', auc)}")
print()

# Detailed hyperparameter analysis
best_params = knn_gs.best_params_
print("=" * 50)
print("HYPERPARAMETER ANALYSIS")
print("=" * 50)
```

```
print("Best k-NN Parameters:")
for param, value in best_params.items():
    print(f"  {param}: {value}")

print(f"\nHyperparameter Impact Analysis:")

# k value analysis
k_val = best_params['n_neighbors']
print(f"k = {k_val}:")
if k_val <= 5:
    print(f"  - Small k: High sensitivity to local patterns")
    print(f"  - Risk: Overfitting to noise, unstable predictions")
    print(f"  - Benefit: Captures fine-grained decision boundaries")
elif k_val <= 15:
    print(f"  - Medium k: Balanced between local and global patterns")
    print(f"  - Good trade-off between bias and variance")
    print(f"  - Reasonable choice for most datasets")
else:
    print(f"  - Large k: Smoothed predictions, more global patterns")
    print(f"  - Risk: Underfitting, overly simplified boundaries")
    print(f"  - Benefit: Stable, less sensitive to outliers")

# Weight analysis
weight_scheme = best_params['weights']
print(f"\nWeights = '{weight_scheme}':")
if weight_scheme == 'uniform':
    print(f"  - All k neighbors vote equally")
    print(f"  - Simple democratic approach")
    print(f"  - Works well when neighbors are relatively close")
else:
    print(f"  - Closer neighbors have more influence (1/distance weighting)")
    print(f"  - Reduces impact of distant neighbors")
    print(f"  - Better for varying neighborhood densities")

# Distance metric analysis
metric = best_params['metric']
p_val = best_params.get('p', 2)
print(f"\nDistance Metric = '{metric}' (p={p_val}):")
if metric == 'euclidean' or (metric == 'minkowski' and p_val == 2):
    print(f"  - Euclidean distance: Standard geometric distance")
    print(f"  - Sensitive to all feature dimensions equally")
    print(f"  - Works well with standardized features")
elif metric == 'manhattan' or (metric == 'minkowski' and p_val == 1):
    print(f"  - Manhattan distance: Sum of absolute differences")
    print(f"  - More robust to outliers than Euclidean")
    print(f"  - Good for high-dimensional spaces")
else:
    print(f"  - Minkowski distance: Generalized distance metric")
    print(f"  - p={p_val} parameter controls distance calculation")
```

```
# Class imbalance impact analysis
print(f"\nClass Imbalance Considerations:")
class_0_ratio = (y_train == 0).mean()
class_1_ratio = (y_train == 1).mean()
print(f"  - Training set: {class_0_ratio:.1%} no diabetes, {class_1_ratio:.1%} diabetes")
print(f"  - k-NN has no built-in class balancing")
print(f"  - With k={k_val}, majority class can dominate neighborhoods")
expected_majority_neighbors = k_val * class_0_ratio
print(f"  - Expected neighbors from majority class: ~{expected_majority_neighbors:.1f} ou

if expected_majority_neighbors > k_val/2:
    print(f"  - Risk: Majority class bias in predictions")
    print(f"  - Mitigation: Distance weighting helps emphasize closer neighbors")
else:
    print(f"  - Balanced neighborhood representation likely")

# Feature space analysis
print(f"\nFeature Space Analysis:")
print(f"  - 3-dimensional feature space (after dimensionality reduction)")
print(f"  - Features standardized (mean=0, std=1)")
print(f"  - Curse of dimensionality: Minimal risk with 3 features")
print(f"  - Training density: {len(X_train)} samples in 3D space")

# Medical interpretation
print(f"\nMedical Decision Making with k-NN:")
print(f"  - Finds {k_val} most similar patients to make diagnosis")
print(f"  - Similarity based on: Glucose, Pregnancies, Diabetes family history")
print(f"  - {'Distance-weighted' if weight_scheme == 'distance' else 'Democratic'} voting
print(f"  - Interpretable: 'Patients like you had diabetes {rec:.1%} of the time'")

# Performance assessment
print(f"\nk-NN Performance Summary:")
print(f"  Test AUC: {auc:.4f} (vs LR baseline: {lr_baseline['auc']:.4f})")
auc_diff = auc - lr_baseline['auc']
if auc_diff > 0.01:
    print(f"  - k-NN outperforms logistic regression (+{auc_diff:.4f})")
    print(f"  - Non-linear patterns in data benefit from instance-based learning")
elif auc_diff < -0.01:
    print(f"  - k-NN underperforms logistic regression ({auc_diff:.4f})")
    print(f"  - Linear relationships captured better by parametric models")
else:
    print(f"  - k-NN performs similarly to logistic regression ({auc_diff:+.4f})")
    print(f"  - Data appears to have mostly linear relationships")

print(f"\nClassification Trade-offs:")
precision_recall_diff = prec - rec
if abs(precision_recall_diff) > 0.05:
    if precision_recall_diff > 0:
        print(f"  - Higher precision ({prec:.4f}) than recall ({rec:.4f})")
        print(f"  - Conservative: Fewer false diabetes predictions")
        print(f"  - Medical impact: Some diabetes cases missed")
```

```
    else:
        print(f"  - Higher recall ({rec:.4f}) than precision ({prec:.4f})")
        print(f"  - Aggressive: Catches more diabetes cases")
        print(f"  - Medical impact: More false alarms, but fewer missed cases")
else:
    print(f"  - Balanced precision ({prec:.4f}) and recall ({rec:.4f})")
    print(f"  - Good trade-off between false positives and false negatives")

recommendation = "Use k-NN" if auc > lr_baseline['auc'] + 0.01 else "Logistic regression
print(f"\nRecommendation: {recommendation}")

# Store results for ensemble comparison
knn_result = {
    "test": {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "auc": auc},
    "estimator": knn_gs,
    "best_params": best_params
}
```

```
============================================================
k-NEAREST NEIGHBORS (k-NN) ANALYSIS
============================================================
Hyperparameter Grid Search Configuration:
- k values: [3, 5, 7, 9, 11, 15, 21, 25] (number of neighbors)
- Weights: ['uniform', 'distance'] (neighbor influence)
- Metrics: ['euclidean', 'manhattan', 'minkowski'] (distance calculation)
- p values: [1, 2] (Minkowski parameter: 1=Manhattan, 2=Euclidean)
- Total combinations: 96

k-NN Algorithm Characteristics:
- Instance-based learning: Uses local patterns in feature space
- Sensitive to feature scaling (our standardization is crucial)
- No explicit model training: 'Lazy learning' approach
- Naturally handles non-linear decision boundaries

Running comprehensive k-NN grid search...
=== k-NN (Comprehensive) — Grid Search CV Results ===
Best CV Score (F1): 0.6124
Best Parameters: {'metric': 'euclidean', 'n_neighbors': 9, 'p': 1, 'weights': 'unifor

=== k-NN (Comprehensive) — Test Set ===
Accuracy : 0.6818
Precision: 0.5581
Recall   : 0.4444
F1-Score : 0.4948
AUC      : 0.7095
Confusion Matrix: TN=81, FP=19, FN=30, TP=24

--- Δ vs Logistic Regression baseline ---
Δ Accuracy : -0.0585
Δ Precision: -0.0548
Δ Recall   : -0.2593
Δ F1-Score : -0.1604
Δ AUC      : -0.0922
```

```
==================================================
HYPERPARAMETER ANALYSIS
==================================================
Best k-NN Parameters:
  metric: euclidean
  n_neighbors: 9
  p: 1
  weights: uniform

Hyperparameter Impact Analysis:
k = 9:
  - Medium k: Balanced between local and global patterns
  - Good trade-off between bias and variance
  - Reasonable choice for most datasets

Weights = 'uniform':
  - All k neighbors vote equally
  - Simple democratic approach
  - Works well when neighbors are relatively close

Distance Metric = 'euclidean' (p=1):
  - Euclidean distance: Standard geometric distance
```

Now we have comprehensively tested k-NN, checking values for multiple different ks and employing the ideal k, which was 9. Our analysis shows that k-NN underperformed logistic regression once again, this time unanimously among metrics. Logistic regression remains the top performing model. This is likely about to change, as we will test two very powerful models next.

Random Forest (Tree-based model)

We use a Random Forest as our required "tree model". We tune the main capacity and regularization knobs: number of trees (`n_estimators`), tree depth (`max_depth`), minimum samples per split/leaf, and `class_weight` to handle class imbalance. We evaluate with 5-fold CV and then on the test set.

```
# Comprehensive Random Forest Implementation
from sklearn.ensemble import RandomForestClassifier
import numpy as np

print("=" * 60)
print("RANDOM FOREST ANALYSIS")
print("=" * 60)

rf = RandomForestClassifier(random_state=RANDOM_STATE)

# Extended hyperparameter grid
```

```
# Extended hyperparameter grid
rf_param_grid = {
    "n_estimators": [100, 300],              # Number of trees
    "max_depth": [None, 10, 20],               # Tree depth control
    "min_samples_split": [2, 10],              # Split requirements
    "min_samples_leaf": [1, 4],                 # Leaf size control
    "max_features": ["sqrt", None],           # Feature sampling
    "class_weight": [None, "balanced"],               # Class imbalance
    "bootstrap": [True]                         # Sampling method
}

print("Hyperparameter Grid Search Configuration:")
print(f"- n_estimators: {rf_param_grid['n_estimators']} (number of trees)")
print(f"- max_depth: {rf_param_grid['max_depth']} (tree depth limit)")
print(f"- min_samples_split: {rf_param_grid['min_samples_split']} (minimum samples to spl
print(f"- min_samples_leaf: {rf_param_grid['min_samples_leaf']} (minimum leaf size)")
print(f"- max_features: {rf_param_grid['max_features']} (features per split)")
print(f"- class_weight: {rf_param_grid['class_weight']} (imbalance handling)")
print(f"- bootstrap: {rf_param_grid['bootstrap']} (sampling method)")
total_combinations = 1
for param_list in rf_param_grid.values():
    total_combinations *= len(param_list)
print(f"- Total combinations: {total_combinations}")
print()

# Grid search
rf_gs = GridSearchCV(
    rf,
    rf_param_grid,
    scoring="f1",
    cv=cv,
    n_jobs=-1
)

print("Running Random Forest grid search...")
rf_gs.fit(X_train, y_train)

# Manual evaluation
print(f"=== Random Forest — Grid Search CV Results ===")
print(f"Best CV Score (F1): {rf_gs.best_score_:.4f}")
print(f"Best Parameters: {rf_gs.best_params_}")
print()

# Test set evaluation
best_rf = rf_gs.best_estimator_
y_pred = best_rf.predict(X_test)
y_prob = best_rf.predict_proba(X_test)[:, 1]

# Calculate test metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
```

```
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()

print(f"=== Random Forest — Test Set ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"AUC      : {auc:.4f}")
print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}\n")

# Comparison to baseline
def delta_str(metric, value):
    base = lr_baseline.get(metric)
    if base is None:
        return "n/a"
    d = value - base
    sign = "+" if d >= 0 else ""
    return f"{sign}{d:.4f}"

print(f"--- Δ vs Logistic Regression baseline ---")
print(f"Δ Accuracy : {delta_str('accuracy', acc)}")
print(f"Δ Precision: {delta_str('precision', prec)}")
print(f"Δ Recall   : {delta_str('recall', rec)}")
print(f"Δ F1-Score : {delta_str('f1', f1)}")
print(f"Δ AUC      : {delta_str('auc', auc)}")
print()

# Key interpretable outputs
best_params = rf_gs.best_params_
print("=" * 50)
print("RANDOM FOREST INTERPRETABILITY ANALYSIS")
print("=" * 50)

# Feature importance analysis
feature_names = ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFunction_l
feature_importance = best_rf.feature_importances_
importance_pairs = list(zip(feature_names, feature_importance))
importance_pairs.sort(key=lambda x: x[1], reverse=True)

print("Feature Importance Ranking:")
for i, (feature, importance) in enumerate(importance_pairs, 1):
    percentage = importance * 100
    print(f"  {i}. {feature}: {importance:.4f} ({percentage:.1f}%)")

# Tree ensemble characteristics
print(f"\nEnsemble Characteristics:")
print(f"  Number of trees: {best_params['n_estimators']}")
print(f"  Max tree depth: {best_params['max_depth']}")
```

```
print(f"  Max tree depth: {best_params['max_depth']}")
print(f"  Min samples per split: {best_params['min_samples_split']}")
print(f"  Min samples per leaf: {best_params['min_samples_leaf']}")
print(f"  Features per split: {best_params['max_features']}")
print(f"  Bootstrap sampling: {best_params['bootstrap']}")
print(f"  Class weighting: {best_params['class_weight']}")

# Model complexity assessment
if best_params['max_depth'] is None:
    depth_analysis = "Unlimited depth - trees can grow until pure leaves"
else:
    depth_analysis = f"Limited to {best_params['max_depth']} levels - controlled complexi

print(f"\nComplexity Analysis:")
print(f"  Tree depth: {depth_analysis}")
print(f"  Leaf control: Min {best_params['min_samples_leaf']} samples per leaf")
print(f"  Split control: Min {best_params['min_samples_split']} samples to split")

# Out-of-bag analysis (if bootstrap=True)
if best_params['bootstrap']:
    try:
        oob_score = best_rf.oob_score_
        print(f"\nOut-of-Bag Analysis:")
        print(f"  OOB Score: {oob_score:.4f}")
        print(f"  Internal validation without separate test set")
    except:
        print(f"\nOut-of-Bag Analysis: Not available (oob_score not enabled)")

# Decision path interpretability
print(f"\nInterpretability Features:")
print(f"  - Feature importance: Shows which variables matter most")
print(f"  - Tree structure: Each tree provides interpretable rules")
print(f"  - Voting mechanism: Democratic decision across {best_params['n_estimators']} tr
print(f"  - Threshold-based: Clear if-then-else decision paths")

# Overfitting assessment
variance_indicator = "High" if best_params['max_depth'] is None and best_params['min_samp
print(f"\nOverfitting Assessment:")
print(f"  Variance risk: {variance_indicator}")
print(f"  Bootstrap aggregation reduces overfitting")
print(f"  Feature randomness adds diversity")

# Medical interpretability
print(f"\nMedical Decision Interpretation:")
top_feature = importance_pairs[0][0]
top_importance = importance_pairs[0][1]
print(f"  Primary decision factor: {top_feature} ({top_importance:.1%} importance)")
print(f"  Ensemble voting: {best_params['n_estimators']} independent medical 'opinions'")
print(f"  Rule-based decisions: Interpretable threshold conditions")

# Performance summary
```

```
print(f"\nRandom Forest Performance Summary:")
print(f"  Test AUC: {auc:.4f} (vs LR baseline: {lr_baseline['auc']:.4f})")
auc_diff = auc - lr_baseline['auc']
if auc_diff > 0.01:
    print(f"  - Random Forest outperforms baseline (+{auc_diff:.4f})")
elif auc_diff < -0.01:
    print(f"  - Random Forest underperforms baseline ({auc_diff:.4f})")
else:
    print(f"  - Random Forest performs similarly to baseline ({auc_diff:+.4f})")

recommendation = "Use Random Forest" if auc > lr_baseline['auc'] + 0.01 else "Logistic re
print(f"\nRecommendation: {recommendation}")

# Store results for ensemble comparison
rf_result = {
    "test": {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "auc": auc},
    "estimator": rf_gs,
    "best_params": best_params,
    "feature_importance": dict(zip(feature_names, feature_importance))
}
```

```
============================================================
RANDOM FOREST ANALYSIS
============================================================
Hyperparameter Grid Search Configuration:
- n_estimators: [100, 300] (number of trees)
- max_depth: [None, 10, 20] (tree depth limit)
- min_samples_split: [2, 10] (minimum samples to split)
- min_samples_leaf: [1, 4] (minimum leaf size)
- max_features: ['sqrt', None] (features per split)
- class_weight: [None, 'balanced'] (imbalance handling)
- bootstrap: [True] (sampling method)
- Total combinations: 96

Running Random Forest grid search...
=== Random Forest — Grid Search CV Results ===
Best CV Score (F1): 0.6528
Best Parameters: {'bootstrap': True, 'class_weight': 'balanced', 'max_depth': None, '

=== Random Forest — Test Set ===
Accuracy : 0.6883
Precision: 0.5577
Recall   : 0.5370
F1-Score : 0.5472
AUC      : 0.7544
Confusion Matrix: TN=77, FP=23, FN=25, TP=29

--- Δ vs Logistic Regression baseline ---
Δ Accuracy : -0.0520
Δ Precision: -0.0552
Δ Recall   : -0.1667
Δ F1-Score : -0.1080
Δ AUC      : -0.0473
```

```
==================================================
RANDOM FOREST INTERPRETABILITY ANALYSIS
==================================================
Feature Importance Ranking:
  1. Glucose_scaled: 0.5731 (57.3%)
  2. DiabetesPedigreeFunction_log_scaled: 0.2982 (29.8%)
  3. Pregnancies_sqrt_scaled: 0.1287 (12.9%)

Ensemble Characteristics:
  Number of trees: 300
  Max tree depth: None
  Min samples per split: 10
  Min samples per leaf: 4
  Features per split: None
  Bootstrap sampling: True
  Class weighting: balanced

Complexity Analysis:
  Tree depth: Unlimited depth - trees can grow until pure leaves
  Leaf control: Min 4 samples per leaf
  Split control: Min 10 samples to split

Out-of-Bag Analysis: Not available (oob_score not enabled)

Interpretability Features:
```

Surprisingly, even Random Forests, the tree-based solution we selected specifically for its effectiveness, did not outperform logistic regression, regardless of hyperparameters.

For our ensemble method, we will be combining the force of all three of our tried competitors using a soft voting system. We will see if, combined together, these three algorithms can work together to outperform logistic regression.

Soft Voting Ensemble (SVM + k-NN + Random Forest)

We build a soft-voting classifier that averages the class probabilities from the **best SVM**, **best k-NN**, and **best Random Forest**. Ensembles often improve robustness by combining models with different biases (margin-based SVM, instance-based k-NN, tree-based RF).

Choosing this as the ensemble method actually gives us a lot of insight into these models. It tells us, are they underperforming for the same reason, or do they have different shortcomings that can be accomadated for? We will be finding out now.

```
# Soft Voting Ensemble - Combining SVM, k-NN, and Random Forest
from sklearn.ensemble import VotingClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.ensemble import RandomForestClassifier

print("=" * 60)
print("SOFT VOTING ENSEMBLE ANALYSIS")
print("=" * 60)

# Recreate individual models with their best hyperparameters
# (Using the best parameters found from previous grid searches)

# Best SVM (from previous results)
best_svm = SVC(
    C=0.1,                    # From SVM grid search
    class_weight='balanced',
    gamma=0.3,
    kernel='rbf',
    probability=True,         # Required for soft voting
    random_state=RANDOM_STATE
)

# Best k-NN (from your grid search results)
best_knn = KNeighborsClassifier(
    n_neighbors=9,
    weights='uniform',
    metric='euclidean',
    p=1
)

# Best Random Forest (from your grid search results)
best_rf = RandomForestClassifier(
    n_estimators=300,
    max_depth=None,
    min_samples_split=10,
    min_samples_leaf=4,
    max_features=None,
    class_weight='balanced',
    bootstrap=True,
    random_state=RANDOM_STATE
)

print("Ensemble Components:")
print("1. SVM (RBF): C=0.1, gamma=0.3, balanced classes")
print("2. k-NN: Update with your actual best parameters")
print("3. Random Forest: Update with your actual best parameters")
print()

# Create soft voting ensemble
soft_voting_ensemble = VotingClassifier(
    estimators=[
        ('svm', best_svm),
        ('knn', best_knn),
        ('rf', best_rf)
```

```
        ('rf', best_rf)
    ],
    voting='soft'  # Uses predicted probabilities for voting
)

print("Soft Voting Mechanism:")
print("- Each model contributes probability estimates")
print("- Final prediction = average of probability predictions")
print("- More sophisticated than hard voting (majority vote)")
print("- Leverages confidence levels from each algorithm")
print()

# Fit ensemble and evaluate
print("Training soft voting ensemble...")
soft_voting_ensemble.fit(X_train, y_train)

# Test set evaluation
y_pred = soft_voting_ensemble.predict(X_test)
y_prob = soft_voting_ensemble.predict_proba(X_test)[:, 1]

# Calculate test metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()

print(f"=== Soft Voting Ensemble — Test Set ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"AUC      : {auc:.4f}")
print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}\n")

# Comparison to baseline and individual models
def delta_str(metric, value):
    base = lr_baseline.get(metric)
    if base is None:
        return "n/a"
    d = value - base
    sign = "+" if d >= 0 else ""
    return f"{sign}{d:.4f}"

print(f"--- Δ vs Logistic Regression baseline ---")
print(f"Δ Accuracy : {delta_str('accuracy', acc)}")
print(f"Δ Precision: {delta_str('precision', prec)}")
print(f"Δ Recall   : {delta_str('recall', rec)}")
print(f"Δ F1-Score : {delta_str('f1', f1)}")
print(f"Δ AUC      : {delta_str('auc', auc)}")
```

```
print()

# Individual model predictions for comparison
print("Individual Model Contributions:")
individual_predictions = {}

# Get individual predictions
for name, model in soft_voting_ensemble.named_estimators_.items():
    pred_proba = model.predict_proba(X_test)[:, 1]
    individual_auc = roc_auc_score(y_test, pred_proba)
    individual_predictions[name] = {
        'probabilities': pred_proba,
        'auc': individual_auc
    }
    print(f"  {name.upper()} AUC: {individual_auc:.4f}")

ensemble_auc = auc
print(f"  ENSEMBLE AUC: {ensemble_auc:.4f}")
print()

# Analyze ensemble effect
print("=" * 50)
print("ENSEMBLE ANALYSIS")
print("=" * 50)

# Check if ensemble improved over best individual
best_individual_auc = max([model['auc'] for model in individual_predictions.values()])
ensemble_improvement = ensemble_auc - best_individual_auc

print(f"Best individual model AUC: {best_individual_auc:.4f}")
print(f"Ensemble AUC: {ensemble_auc:.4f}")
print(f"Ensemble improvement: {ensemble_improvement:+.4f}")

if ensemble_improvement > 0.005:
    print("✓ Ensemble successfully combines model strengths")
elif ensemble_improvement < -0.005:
    print("✗ Ensemble performs worse than best individual model")
else:
    print("~ Ensemble performs similarly to best individual model")

# Diversity analysis
print(f"\nModel Diversity Analysis:")
svm_probs = individual_predictions['svm']['probabilities']
knn_probs = individual_predictions['knn']['probabilities']
rf_probs = individual_predictions['rf']['probabilities']

# Calculate pairwise correlations
svm_knn_corr = np.corrcoef(svm_probs, knn_probs)[0,1]
svm_rf_corr = np.corrcoef(svm_probs, rf_probs)[0,1]
knn_rf_corr = np.corrcoef(knn_probs, rf_probs)[0,1]
```

```
print(f"  SVM ↔ k-NN correlation: {svm_knn_corr:.3f}")
print(f"  SVM ↔ RF correlation: {svm_rf_corr:.3f}")
print(f"  k-NN ↔ RF correlation: {knn_rf_corr:.3f}")

avg_correlation = (svm_knn_corr + svm_rf_corr + knn_rf_corr) / 3
print(f"  Average correlation: {avg_correlation:.3f}")

if avg_correlation > 0.8:
    print("  High correlation - models make similar predictions")
    print("  Limited benefit from ensemble diversity")
elif avg_correlation < 0.6:
    print("  Low correlation - good model diversity")
    print("  Ensemble can leverage different model strengths")
else:
    print("  Moderate correlation - reasonable ensemble diversity")

# Final recommendation
print(f"\nEnsemble Performance Summary:")
lr_vs_ensemble = ensemble_auc - lr_baseline['auc']
print(f"  Ensemble AUC: {ensemble_auc:.4f}")
print(f"  vs Logistic Regression: {lr_vs_ensemble:+.4f}")

if lr_vs_ensemble > 0.01:
    recommendation = "Use ensemble model"
    print("  ✓ Ensemble outperforms simple logistic regression")
elif lr_vs_ensemble < -0.01:
    recommendation = "Use logistic regression"
    print("  ✗ Ensemble underperforms compared to logistic regression")
else:
    recommendation = "Logistic regression remains competitive"
    print("  ~ Ensemble performs similarly to logistic regression")

print(f"\nFinal Recommendation: {recommendation}")

# Store results
ensemble_result = {
    "test": {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "auc": auc},
    "individual_aucs": {name: model['auc'] for name, model in individual_predictions.item
    "ensemble_improvement": ensemble_improvement,
    "model_correlations": {"svm_knn": svm_knn_corr, "svm_rf": svm_rf_corr, "knn_rf": knn_
}
```

```
============================================================
SOFT VOTING ENSEMBLE ANALYSIS
============================================================
Ensemble Components:
1. SVM (RBF): C=0.1, gamma=0.3, balanced classes
2. k-NN: Update with your actual best parameters
3. Random Forest: Update with your actual best parameters

Soft Voting Mechanism:
```

```
- Each model contributes probability estimates
- Final prediction = average of probability predictions
- More sophisticated than hard voting (majority vote)
- Leverages confidence levels from each algorithm

Training soft voting ensemble...
=== Soft Voting Ensemble — Test Set ===
Accuracy : 0.7338
Precision: 0.6444
Recall   : 0.5370
F1-Score : 0.5859
AUC      : 0.7617
Confusion Matrix: TN=84, FP=16, FN=25, TP=29

--- Δ vs Logistic Regression baseline ---
Δ Accuracy : -0.0065
Δ Precision: +0.0315
Δ Recall   : -0.1667
Δ F1-Score : -0.0693
Δ AUC      : -0.0400

Individual Model Contributions:
  SVM AUC: 0.7878
  KNN AUC: 0.7095
  RF AUC: 0.7544
  ENSEMBLE AUC: 0.7617

==================================================
ENSEMBLE ANALYSIS
==================================================
Best individual model AUC: 0.7878
Ensemble AUC: 0.7617
Ensemble improvement: -0.0261
✗ Ensemble performs worse than best individual model

Model Diversity Analysis:
  SVM ↔ k-NN correlation: 0.855
  SVM ↔ RF correlation: 0.892
  k-NN ↔ RF correlation: 0.875
  Average correlation: 0.874
  High correlation - models make similar predictions
  Limited benefit from ensemble diversity

Ensemble Performance Summary:
  Ensemble AUC: 0.7617
  vs Logistic Regression: -0.0400
  ✗ Ensemble underperforms compared to logistic regression

Final Recommendation: Use logistic regression
```

The models were highly coordinated, and therefore their shortcomings all combined and once again the model failed to perform. Logistic regression may be the best model for this dataset.

We do not want to give up on ensemble methods yet. Let's give it one more attempt, with

Gradient Boosting.

```
# XGBoost Gradient Boosting Implementation
from xgboost import XGBClassifier
import numpy as np

print("=" * 60)
print("XGBOOST GRADIENT BOOSTING ANALYSIS")
print("=" * 60)

# XGBoost classifier
xgb = XGBClassifier(
    random_state=RANDOM_STATE,
    eval_metric='logloss'  # Suppress warning
)

# Comprehensive XGBoost parameter grid
xgb_param_grid = {
    "n_estimators": [100, 200, 300],           # Number of boosting rounds
    "max_depth": [3, 4, 6, 8],                 # Tree depth
    "learning_rate": [0.01, 0.1, 0.2],        # Step size shrinkage
    "subsample": [0.8, 1.0],                   # Sample ratio per tree
    "colsample_bytree": [0.8, 1.0],           # Feature sampling per tree
    "reg_alpha": [0, 0.1],                     # L1 regularization
    "reg_lambda": [1, 1.5],                    # L2 regularization
    "scale_pos_weight": [1, 1.87]              # Class imbalance handling (ratio = 400/214
}

print("XGBoost Hyperparameter Grid:")
print(f"- n_estimators: {xgb_param_grid['n_estimators']} (boosting rounds)")
print(f"- max_depth: {xgb_param_grid['max_depth']} (tree complexity)")
print(f"- learning_rate: {xgb_param_grid['learning_rate']} (step size)")
print(f"- subsample: {xgb_param_grid['subsample']} (row sampling)")
print(f"- colsample_bytree: {xgb_param_grid['colsample_bytree']} (feature sampling)")
print(f"- reg_alpha: {xgb_param_grid['reg_alpha']} (L1 regularization)")
print(f"- reg_lambda: {xgb_param_grid['reg_lambda']} (L2 regularization)")
print(f"- scale_pos_weight: {xgb_param_grid['scale_pos_weight']} (class imbalance)")

total_combinations = 1
for param_list in xgb_param_grid.values():
    total_combinations *= len(param_list)
print(f"- Total combinations: {total_combinations}")
print()

print("XGBoost Algorithm Characteristics:")
print("- Sequential boosting: Each tree corrects previous tree errors")
print("- Gradient-based optimization: Minimizes prediction residuals")
print("- Built-in regularization: Controls overfitting")
print("- Handles class imbalance: scale_pos_weight parameter")
print()
```

```
print()

# Grid search
xgb_gs = GridSearchCV(
    xgb,
    xgb_param_grid,
    scoring="f1",
    cv=cv,
    n_jobs=-1
)

print("Running XGBoost grid search...")
xgb_gs.fit(X_train, y_train)

# Manual evaluation
print(f"=== XGBoost — Grid Search CV Results ===")
print(f"Best CV Score (F1): {xgb_gs.best_score_:.4f}")
print(f"Best Parameters: {xgb_gs.best_params_}")
print()

# Test set evaluation
best_xgb = xgb_gs.best_estimator_
y_pred = best_xgb.predict(X_test)
y_prob = best_xgb.predict_proba(X_test)[:, 1]

# Calculate test metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()

print(f"=== XGBoost — Test Set ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"AUC      : {auc:.4f}")
print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}\n")

# Comparison to baseline and other models
def delta_str(metric, value):
    base = lr_baseline.get(metric)
    if base is None:
        return "n/a"
    d = value - base
    sign = "+" if d >= 0 else ""
    return f"{sign}{d:.4f}"

print(f"--- Δ vs Logistic Regression baseline ---")
```

```
print(f"Δ Accuracy : {delta_str('accuracy', acc)}")
print(f"Δ Precision: {delta_str('precision', prec)}")
print(f"Δ Recall   : {delta_str('recall', rec)}")
print(f"Δ F1-Score : {delta_str('f1', f1)}")
print(f"Δ AUC      : {delta_str('auc', auc)}")
print()

# XGBoost-specific analysis
best_params = xgb_gs.best_params_
print("=" * 50)
print("XGBOOST INTERPRETABILITY ANALYSIS")
print("=" * 50)

# Feature importance analysis
feature_names = ['Glucose_scaled', 'Pregnancies_sqrt_scaled', 'DiabetesPedigreeFunction_l
feature_importance = best_xgb.feature_importances_
importance_pairs = list(zip(feature_names, feature_importance))
importance_pairs.sort(key=lambda x: x[1], reverse=True)

print("Feature Importance Ranking:")
for i, (feature, importance) in enumerate(importance_pairs, 1):
    percentage = importance * 100
    print(f"  {i}. {feature}: {importance:.4f} ({percentage:.1f}%)")

# Boosting characteristics
print(f"\nBoosting Configuration:")
print(f"  Number of estimators: {best_params['n_estimators']}")
print(f"  Learning rate: {best_params['learning_rate']}")
print(f"  Max tree depth: {best_params['max_depth']}")
print(f"  Row sampling: {best_params['subsample']}")
print(f"  Feature sampling: {best_params['colsample_bytree']}")
print(f"  L1 regularization (alpha): {best_params['reg_alpha']}")
print(f"  L2 regularization (lambda): {best_params['reg_lambda']}")
print(f"  Class weight scaling: {best_params['scale_pos_weight']}")

# Model complexity assessment
complexity_score = 0
if best_params['learning_rate'] <= 0.1:
    complexity_score += 1  # Conservative learning
if best_params['max_depth'] <= 4:
    complexity_score += 1  # Shallow trees
if best_params['reg_alpha'] > 0 or best_params['reg_lambda'] > 1:
    complexity_score += 1  # Regularization active

if complexity_score >= 2:
    complexity_assessment = "Conservative (low overfitting risk)"
elif complexity_score == 1:
    complexity_assessment = "Moderate (balanced complexity)"
else:
    complexity_assessment = "Aggressive (higher overfitting risk)"
```

```
print(f"\nModel Complexity Assessment: {complexity_assessment}")

# Gradient boosting insights
print(f"\nGradient Boosting Insights:")
if best_params['learning_rate'] <= 0.1:
    print(f"  - Low learning rate: Gradual error correction, stable convergence")
else:
    print(f"  - High learning rate: Faster learning, risk of overshooting")

if best_params['max_depth'] <= 4:
    print(f"  - Shallow trees: Focus on main effects, interpretable splits")
else:
    print(f"  - Deep trees: Captures complex interactions, higher variance")

print(f"  - Sequential learning: Each tree targets residual errors")
print(f"  - Built-in cross-validation: Early stopping prevents overfitting")

# Comparison to other ensemble methods
print(f"\nEnsemble Method Comparison:")
print(f"  XGBoost AUC: {auc:.4f}")
print(f"  vs Random Forest: {auc - 0.7544:+.4f}")  # Using your RF result
print(f"  vs Soft Voting: {auc - 0.7617:+.4f}")   # Using ensemble result
print(f"  vs Logistic Regression: {auc - lr_baseline['auc']:+.4f}")

# Final assessment
auc_improvement = auc - lr_baseline['auc']
if auc_improvement > 0.02:
    assessment = "XGBoost significantly outperforms baseline"
    recommendation = "Use XGBoost for optimal performance"
elif auc_improvement > 0.01:
    assessment = "XGBoost shows meaningful improvement"
    recommendation = "Consider XGBoost vs logistic regression trade-offs"
elif auc_improvement > -0.01:
    assessment = "XGBoost performs similarly to baseline"
    recommendation = "Logistic regression remains competitive"
else:
    assessment = "XGBoost underperforms baseline"
    recommendation = "Use logistic regression"

print(f"\nPerformance Assessment: {assessment}")
print(f"Final Recommendation: {recommendation}")

# Medical interpretation
print(f"\nMedical Decision Support:")
top_feature = importance_pairs[0][0]
top_importance = importance_pairs[0][1]
print(f"  Primary decision factor: {top_feature} ({top_importance:.1%} importance)")
print(f"  Sequential refinement: {best_params['n_estimators']} decision trees")
print(f"  Error correction: Each tree improves on previous predictions")
print(f"  Confidence calibration: Gradient-based probability estimates")
```

```
# Store results
xgb_result = {
    "test": {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "auc": auc},
    "best_params": best_params,
    "feature_importance": dict(zip(feature_names, feature_importance)),
    "improvement_over_lr": auc_improvement
}
```

```
============================================================
XGBOOST GRADIENT BOOSTING ANALYSIS
============================================================
XGBoost Hyperparameter Grid:
- n_estimators: [100, 200, 300] (boosting rounds)
- max_depth: [3, 4, 6, 8] (tree complexity)
- learning_rate: [0.01, 0.1, 0.2] (step size)
- subsample: [0.8, 1.0] (row sampling)
- colsample_bytree: [0.8, 1.0] (feature sampling)
- reg_alpha: [0, 0.1] (L1 regularization)
- reg_lambda: [1, 1.5] (L2 regularization)
- scale_pos_weight: [1, 1.87] (class imbalance)
- Total combinations: 1152

XGBoost Algorithm Characteristics:
- Sequential boosting: Each tree corrects previous tree errors
- Gradient-based optimization: Minimizes prediction residuals
- Built-in regularization: Controls overfitting
- Handles class imbalance: scale_pos_weight parameter

Running XGBoost grid search...
=== XGBoost — Grid Search CV Results ===
Best CV Score (F1): 0.6578
Best Parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.01, 'max_depth': 6, 'n_

=== XGBoost — Test Set ===
Accuracy : 0.7078
Precision: 0.5818
Recall   : 0.5926
F1-Score : 0.5872
AUC      : 0.7724
Confusion Matrix: TN=77, FP=23, FN=22, TP=32

--- Δ vs Logistic Regression baseline ---
Δ Accuracy : -0.0325
Δ Precision: -0.0311
Δ Recall   : -0.1111
Δ F1-Score : -0.0680
Δ AUC      : -0.0293

==================================================
XGBOOST INTERPRETABILITY ANALYSIS
==================================================
Feature Importance Ranking:
  1. Glucose_scaled: 0.5407 (54.1%)
  2. DiabetesPedigreeFunction_log_scaled: 0.2322 (23.2%)
```

```
    3. Pregnancies_sqrt_scaled: 0.2272 (22.7%)

  Boosting Configuration:
    Number of estimators: 200
    Learning rate: 0.01
    Max tree depth: 6
    Row sampling: 0.8
    Feature sampling: 0.8
    L1 regularization (alpha): 0
    L2 regularization (lambda): 1
    Class weight scaling: 1.87
```

```
Start coding or generate with AI.
```

Even XGBoost could not outperform logistic regression.

Method Decisions & Rationale, Final Summary

This project was a comprehensive experience, full of decisions and tests to ensure the best possible model. We experimented with many different ways to enhance the effectiveness and predictability of the dataset, from transitions to feature engineering. We have experimented with many different models, each with many different hyperparameters.

We started off with a large dataset, 768 data points that extend with 8 features and a target variable. The data was messy, unreliable, and even redundant. It was not yet fit for a model. We analyzed it and found these issues. We checked the confusian matrix, searching for null data, and didn't find any.

We then checked the histograms of each feature given the target variable, where we found a lot of new information about how we must tweak the data, and this is where decisions first started to pour in. For the first time, we saw glucose level correlated the most with whether or not a patient had diabetes. We saw insulin, age, and pregnancies correlated too. We found many features had a shift to the right, so we applied log and sqrt functions depending on the severity. We found that a lot of data had impossible 0s — values stored in place of a null — and we systematically removed them.

At this point, our data looked much more normalized, and it was time to start looking into how workable the features are. So, we generated a heat matrix that communicates the correlation between all our different features. We found a strong correlation between insulin and glucose, age and pregnancies. Once again, we were presented with a decision: remove the redundancies, or try to make a ratio out of them. First, we tried to employ a ratio. Strangely, correlation to the target variable saw significant decline, so we removed that step and instead

settled for removing the redundancies. For now, we just removed the redundant skin thickness feature. We would explore removing others later, when we do a VIF analysis.

Our next issue to tackle was outliers. We noticed most features had less than 2% outliers, managable by most models without having to alter the data, but insulin had a whopping 7% outlier count. We had to act on this immediately. It was time to make another decision. After a bunch of research on our options, we settled for winsorization. This basically replaces the outlier values with more reasonable values, without losing data. We ran winsorization, and we now had 0% outliers on insulin. Outliers were now manageable.

We weren't done analyzing the data yet. There was still plenty more to look at, starting with pair plots. Pair plots confirmed our theories, that our correlated datasets were seperable and will likely be very telling for a model. Our lowest correlator, bloodpressure, was not separable. It was a mess — and it would likely be more trouble than it's worth. We still wouldn't remove it quite yet, as it might still have some use if everything else works out.

Our next step was chi-squared tests, where we gained a lot of additional insight. Each feature showed correlation with the target variable. Each feature also had a p value less than .001, so they were seemingly independent. Each feature seemed appropriate to move forward.

This was until VIF analysis, where we had shocking results. Where a VIF score of 10 implies severe multicolinearity issues, we found numbers as high as 76.57 with insulin. Insulin, age, blood pressure, BMI, and glucose all suffered from severe multicolinearity. We could not move forward with all these features. Decisions had to be made. Blood pressure was the easiest to remove, every test showed that it was a poor tell of diabetic state. In order to address this, we got rid of insulin, age, blood pressure, and BMI — our worst offenders. We were left with our three most reliable features: glucose, pregnancies, and diabetes pedigree. We would eventually try reintroducing these features to make sure removing them was the best choice.

Our features weren't perfect yet, however. We still had an issue of missing values, and too many for it to be worthwhile to just remove all rows with missing values. This is where k-nn imputation came in. We imputated in for missing values, based on the nearest neighbor of each point. We also tried mean imputation, but it k-nn imputation had a better preservation of distribution.

We still needed to standardize. Our data was in vastly different ranges, and that meant that each data point was not equally representative. Glucose had a range of 155, pregnancies had a range of 4.22, and diabetes pedigree had a range of 1.15. Once we standardized, our data was finally ready to be implemented into the first model — logistic regression.

We tried a wide range of hyperparameters for logistic regression; 10 different variations, to be exact. The one that performed the best was one that addressed the class imbalance, with a balanced hyperparameter. It was our 6th configuration. It showed that we did have an

balanced hyperparameter. It was our 6th configuration. It showed that we did have an imbalance issue — one we should try addressing other ways before continuing.

So we looked at two options: SMOTE, and random undersampling. We expected SMOTE to perform, but in hindsight, our dataset didn't meet the properties that make SMOTE ideal, such as a large class imbalance (up to 1:10), or a large dataset. We were looking at a dataset of around 800, with 1.87:1 ratio, and our relationships were mostly linear. Our original model, using a balanced hyperparameter, actually performed the best. Random undersampling performed the second best, and SMOTE performed the worst, likely just introducing noise and unrealistic data. Still, for the sake of the project, we left the dataset SMOTE-balanced. It was not a significant difference either way.

Next, we analyzed the ROC curve and the AUC. We got an AUC of .802, representing effective performance where, if a diabetic patient and a nondiabetic patient are randomly selected, there is an 80% chance that the diabetic patient will be ranked higher. This is clinically significant, and shows our model is performing. The ROC curve demonstrated that we could accomplish anywhere from 80% sensitivity with 20% false positive, to 95% sensitivity with 60% false positive. False positives are definitely better than false negatives, as a false negative can endanger ones health, so high sensitivity is good.

Now that we had a decent model going, it was time to loop back to feature engineering to make sure we can not improve our model in any reasonable way. Our first test was to see if we were correct in removing insulin. It was a powerful indicator, it just suffered gravely from multicolinearity. So, we added back in insulin, and ran the model again. The difference was miniscule — definitely not enough to add a dimension of complexity to our model, so we decided to leave it out.

We decided to check if it was worth swapping out pregnancies for age. It sounds more general to go with age, so perhaps age was the better. It turned out not to be true. The model performed much worse with this swap, so we could conclude that pregnancies was the more telling feature.

This presses another question: Is it worth it to remove these features, or can we get more information making a ratio out of them. So we made a glucose insulin ratio, took the logarithm of that, and tested it out as a feature. Once again, the model performed worse across the metrics, showing what we had initially was superior.

Model Evaluation Report

After preparing and refining the dataset, we evaluated a range of models. Each model was carefully tuned using cross-validation and GridSearch, and performance was compared on the test set. The following sections summarize the models, their configurations, and the

insights gained.

The first alternative to Logistic Regression was a Support Vector Machine. An RBF kernel was applied to capture potential non-linear patterns that Logistic Regression might miss. Hyperparameters including C, gamma, kernel type, and class weighting were tuned using GridSearchCV with five-fold stratified cross-validation. The best configuration was C equal to 0.1, gamma equal to 0.3, an RBF kernel, and class weight set to balanced. On the test set, the SVM achieved an accuracy of 0.7143, precision of 0.5735, recall of 0.7222, F1-score of 0.6393, and an AUC of 0.7878. The SVM provided higher recall compared to Logistic Regression, meaning it identified more diabetic patients, but it underperformed overall. The large number of support vectors also suggested that the model was overly complex without stronger generalization.

We next evaluated k-Nearest Neighbors, a simple instance-based learner that predicts outcomes based on the closest observations. Hyperparameters for the number of neighbors, weights, and distance metrics were tuned using GridSearchCV with five-fold stratified cross-validation. The best configuration was nine neighbors with Euclidean distance and uniform weights. On the test set, k-NN achieved an accuracy of 0.6818, precision of 0.5581, recall of 0.4444, F1-score of 0.4948, and an AUC of 0.7095. This model performed the worst overall. It struggled with class imbalance, as it lacks mechanisms to address this issue, and it was highly sensitive to noisy or sparse data even after scaling.

We then applied Random Forest, an ensemble of decision trees designed to improve robustness and capture non-linear relationships. Parameters tuned included depth, leaf size, split rules, number of trees, and class weighting. The best configuration was 300 trees, unrestricted maximum depth, a minimum of four samples per leaf, and balanced class weights. On the test set, the Random Forest achieved an accuracy of 0.6883, precision of 0.5577, recall of 0.5370, F1-score of 0.5472, and an AUC of 0.7544. Despite its complexity, Random Forest did not outperform Logistic Regression. Feature importance analysis showed that Glucose was by far the most influential predictor, while the other features contributed much less.

To test whether combining weaker models could improve performance, we built a soft voting classifier using SVM, k-NN, and Random Forest. The ensemble achieved an accuracy of 0.7338, precision of 0.6444, recall of 0.5370, F1-score of 0.5859, and an AUC of 0.7617. The ensemble improved slightly compared to Random Forest and k-NN individually, but it still did not outperform Logistic Regression. The high correlation between model predictions reduced the potential benefits of ensembling.

We also tested XGBoost, a gradient boosting method known for handling class imbalance and delivering strong performance on structured datasets. Hyperparameters tuned included depth, learning rate, number of estimators, subsampling, and regularization. The best

configuration used 200 estimators, a maximum depth of 6, a learning rate of 0.01, subsampling of 0.8, and scale position weight of 1.87. On the test set, XGBoost achieved an accuracy of 0.7078, precision of 0.5818, recall of 0.5926, F1-score of 0.5872, and an AUC of 0.7724. Although it performed better than some models, XGBoost still trailed behind Logistic Regression in both F1-score and AUC. Feature importance once again highlighted Glucose as the most dominant predictor.

Among all the models tested, Logistic Regression with balanced class weights demonstrated the strongest performance. It achieved the highest F1-score of 0.6552 and the highest AUC of 0.8017, both of which are especially important when evaluating imbalanced medical datasets. Logistic Regression also outperformed more complex approaches, many of which showed tendencies toward overfitting or produced correlated errors. These findings suggest that the relationships in the dataset are largely linear, making Logistic Regression the most suitable choice. Despite extensive experimentation with advanced techniques, the simpler model ultimately provided the best combination of accuracy, sensitivity, and interpretability.