**Data Mining Implementation Project - Pt 3 of 3**

```
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.metrics import (precision_recall_fscore_support, confusion_matrix,
                             roc_curve, auc, classification_report, roc_auc_score,
                             precision_recall_curve)
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
import xgboost as xgb
import warnings
warnings.filterwarnings('ignore')

def print_time_taken(start_time, operation_name):
    """Utility function to print execution time"""
    time_taken = time.time() - start_time
    print(f"Time taken for {operation_name}: {time_taken:.2f} seconds")

def plot_roc_curve(y_test, y_pred_proba, model_name):
    """Plot ROC curve for model evaluation"""
    fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2,
             label=f'ROC curve (area = {roc_auc:.3f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve - {model_name}')
    plt.legend(loc="lower right")
    return plt

def plot_precision_recall_curve(y_test, y_pred_proba, model_name):
    """Plot Precision-Recall curve for model evaluation"""
    precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)
    pr_auc = auc(recall, precision)

    plt.figure(figsize=(8, 6))
    plt.plot(recall, precision, color='blue', lw=2,
             label=f'PR curve (area = {pr_auc:.3f})')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])
    plt.title(f'Precision-Recall Curve - {model_name}')
    plt.legend(loc="lower left")
    return plt

def evaluate_model(model, X_val_scaled, y_val, model_name, threshold=0.5):
    """Comprehensive model evaluation with detailed metrics and adjustable threshold"""
    start_time = time.time()

    # Calculate predictions and probabilities
    y_pred_proba = model.predict_proba(X_val_scaled)[:,1]
    y_pred = (y_pred_proba >= threshold).astype(int)

    # Calculate metrics
    precision, recall, f1, _ = precision_recall_fscore_support(y_val, y_pred, average='binary')
    auc_score = roc_auc_score(y_val, y_pred_proba)
    cm = confusion_matrix(y_val, y_pred)

    # Print detailed results
    print(f"\n{model_name} Results (threshold = {threshold}):")
    print(f"* AUC Score: {auc_score:.3f}")
    print(f"* Precision: {precision:.3f}")
```

```
        print(f"* Recall: {recall:.3f}")
        print(f"* F1-Score: {f1:.3f}")

        print("\nConfusion Matrix:")
        print(cm)

        print("\nClassification Report:")
        print(classification_report(y_val, y_pred))

        # Create visualization
        roc_plot = plot_roc_curve(y_val, y_pred_proba, model_name)
        pr_plot = plot_precision_recall_curve(y_val, y_pred_proba, model_name)

        print_time_taken(start_time, f"evaluating {model_name}")

        return {
            'precision': precision,
            'recall': recall,
            'f1': f1,
            'auc': auc_score,
            'confusion_matrix': cm,
            'threshold': threshold,
            'roc_plot': roc_plot,
            'pr_plot': pr_plot,
            'y_pred_proba': y_pred_proba
        }

    def find_optimal_threshold(y_val, y_pred_proba):
        """Find optimal threshold to balance precision and recall"""
        thresholds = np.arange(0.1, 0.9, 0.05)
        f1_scores = []

        for threshold in thresholds:
            y_pred = (y_pred_proba >= threshold).astype(int)
            _, _, f1, _ = precision_recall_fscore_support(y_val, y_pred, average='binary')
            f1_scores.append(f1)

        optimal_idx = np.argmax(f1_scores)
        return thresholds[optimal_idx]

    def feature_importance_analysis(model, feature_names, model_name):
        """Analyze and visualize feature importances"""
        if hasattr(model, 'feature_importances_'):
            importances = model.feature_importances_
            indices = np.argsort(importances)[::-1]

            plt.figure(figsize=(10, 6))
            plt.title(f'Feature Importances - {model_name}')
            plt.bar(range(len(indices)), importances[indices], align='center')
            plt.xticks(range(len(indices)), [feature_names[i] for i in indices], rotation=90)
            plt.tight_layout()
            return plt
        else:
            print(f"Model {model_name} does not support feature importance analysis")
            return None

    def main():
        # Initial setup
        total_start_time = time.time()
        print("\n=== Hospital Readmission Prediction Model ===")
        print("\nLoading data...")
        start_time = time.time()

        try:
            df = pd.read_csv('diabetic_data.csv')
        except FileNotFoundError:
            print("Looking for dataset in alternative locations...")
            try:
                df = pd.read_csv('/content/diabetic_data.csv')  # For Google Colab
            except FileNotFoundError:
                raise FileNotFoundError("Could not find the diabetic_data.csv file. Please check the file path.")

        print_time_taken(start_time, "loading data")

        # Data exploration
        print("\nExploring data and selecting features...")
        print(f"Dataset shape: {df.shape}")
        missing_values = df.replace('?', np.nan).isnull().sum()
```

```
missing_values = df.replace('?', np.nan).isnull().sum()
print(f"Features with missing values:")
print(missing_values[missing_values > 0])

# Feature selection
essential_features = [
    'race', 'gender', 'age', 'admission_type_id', 'discharge_disposition_id',
    'admission_source_id', 'time_in_hospital', 'num_lab_procedures',
    'num_procedures', 'num_medications', 'number_outpatient',
    'number_emergency', 'number_inpatient', 'number_diagnoses',
    'max_glu_serum', 'A1Cresult', 'diabetesMed', 'readmitted'
]
df = df[essential_features].copy()

# Preprocessing
print("\nPreprocessing data...")
start_time = time.time()

df.replace('?', np.nan, inplace=True)

# Handle missing categorical data with most frequent values
for col in df.select_dtypes(include=['object']).columns:
    if col != 'readmitted':
        df[col] = df[col].fillna(df[col].mode()[0])
        le = LabelEncoder()
        df[col] = le.fit_transform(df[col].astype(str))

# Convert target to binary classification
df['readmitted'] = (df['readmitted'] == '<30').astype(int)

# Display class distribution
print("\nClass distribution:")
print(df['readmitted'].value_counts())
print(f"Positive class percentage: {df['readmitted'].mean()*100:.2f}%")
print("This is a highly imbalanced dataset")
print_time_taken(start_time, "preprocessing")

# Split data
print("\nSplitting data...")
start_time = time.time()
X = df.drop('readmitted', axis=1)
y = df['readmitted']
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)

print(f"Training set: {X_train.shape[0]} samples (positive: {y_train.sum()}, {y_train.mean()*100:.2f}%)")
print(f"Validation set: {X_val.shape[0]} samples (positive: {y_val.sum()}, {y_val.mean()*100:.2f}%)")
print(f"Test set: {X_test.shape[0]} samples (positive: {y_test.sum()}, {y_test.mean()*100:.2f}%)")

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
print_time_taken(start_time, "splitting data")

# Apply SMOTE
print("\nApplying SMOTE for handling class imbalance...")
start_time = time.time()
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train_scaled, y_train)
print(f"After SMOTE - Training samples: {X_train_balanced.shape[0]}")
print(f"Class distribution after SMOTE: {np.bincount(y_train_balanced)}")
print_time_taken(start_time, "SMOTE")

# Define models with optimized hyperparameters
print("\nDefining optimized models...")
models = {
    'XGBoost': xgb.XGBClassifier(
        objective='binary:logistic',
        eval_metric='auc',
        learning_rate=0.02,
        max_depth=5,
        n_estimators=300,
        subsample=0.8,
        colsample_bytree=0.8,
        min_child_weight=3,
        gamma=0.1,
```

```
        scale_pos_weight=5,
        reg_alpha=0.1,
        reg_lambda=1.0,
        random_state=42
    ),
    'Gradient Boosting': GradientBoostingClassifier(
        n_estimators=250,
        learning_rate=0.03,
        max_depth=4,
        min_samples_split=15,
        min_samples_leaf=5,
        subsample=0.8,
        max_features=0.8,
        random_state=42
    ),
    'Random Forest': RandomForestClassifier(
        n_estimators=200,
        max_depth=10,
        min_samples_split=10,
        min_samples_leaf=4,
        max_features='sqrt',
        bootstrap=True,
        class_weight='balanced',
        random_state=42
    )
}

# Train and evaluate models
results = {}
for name, model in models.items():
    print(f"\nTraining {name}...")
    start_time = time.time()
    model.fit(X_train_balanced, y_train_balanced)
    print_time_taken(start_time, f"training {name}")

    # Initial evaluation
    initial_results = evaluate_model(model, X_val_scaled, y_val, name)

    # Find optimal threshold
    print(f"\nFinding optimal threshold for {name}...")
    optimal_threshold = find_optimal_threshold(y_val, initial_results['y_pred_proba'])
    print(f"Optimal threshold: {optimal_threshold:.2f}")

    # Re-evaluate with optimal threshold
    if optimal_threshold != 0.5:
        optimized_results = evaluate_model(model, X_val_scaled, y_val, name, optimal_threshold)
        results[name] = optimized_results
    else:
        results[name] = initial_results

    # Feature importance analysis
    feature_imp_plot = feature_importance_analysis(model, X.columns, name)
    if feature_imp_plot:
        results[name]['feature_importance'] = feature_imp_plot

# Model comparison
print("\nModel Comparison:")
comparison_df = pd.DataFrame({
    'Model': list(results.keys()),
    'Precision': [results[m]['precision'] for m in results],
    'Recall': [results[m]['recall'] for m in results],
    'F1-Score': [results[m]['f1'] for m in results],
    'AUC': [results[m]['auc'] for m in results],
    'Threshold': [results[m]['threshold'] for m in results]
})
print(comparison_df)

# Final evaluation on test set
print("\n=== Final Model Evaluation on Test Set ===")
best_model_name = comparison_df.loc[comparison_df['F1-Score'].idxmax()]['Model']
best_model = models[best_model_name]
best_threshold = results[best_model_name]['threshold']

print(f"\nBest model: {best_model_name} (threshold = {best_threshold:.2f})")
final_results = evaluate_model(best_model, X_test_scaled, y_test, f"{best_model_name} (Test Set)", best_threshold)

# Analysis summary
```

```
    print("\n=== Final Analysis ===")
    print(f"1. Best performing model: {best_model_name}")
    print(f"   * F1-Score: {final_results['f1']:.3f}")
    print(f"   * AUC: {final_results['auc']:.3f}")
    print(f"   * Precision: {final_results['precision']:.3f}")
    print(f"   * Recall: {final_results['recall']:.3f}")
    print("2. Model performance interpretation:")
    print(f"   * The model correctly identifies {final_results['recall']*100:.1f}% of patients who will be readmitted")
    print(f"   * When the model predicts readmission, it is correct {final_results['precision']*100:.1f}% of the time")
    print("3. Key findings:")
    print("   * Handling class imbalance with SMOTE improved model performance")
    print("   * Threshold optimization was crucial for balancing precision and recall")
    print("   * Feature selection and engineering played a significant role")

    print(f"\nTotal execution time: {time.time() - total_start_time:.2f} seconds")

    return {
        'models': models,
        'best_model': best_model,
        'best_model_name': best_model_name,
        'results': results,
        'final_results': final_results,
        'comparison': comparison_df,
        'X_columns': X.columns,
        'scaler': scaler
    }

if __name__ == "__main__":
    main()
```

```
=== Hospital Readmission Prediction Model ===

Loading data...
Time taken for loading data: 0.53 seconds

Exploring data and selecting features...
Dataset shape: (101766, 50)
Features with missing values:
race                 2273
weight              98569
payer_code          40256
medical_specialty   49949
diag_1                 21
diag_2                358
diag_3               1423
max_glu_serum       96420
A1Cresult           84748
dtype: int64

Preprocessing data...

Class distribution:
readmitted
0    90409
1    11357
Name: count, dtype: int64
Positive class percentage: 11.16%
This is a highly imbalanced dataset
Time taken for preprocessing: 0.19 seconds

Splitting data...
Training set: 71236 samples (positive: 7950, 11.16%)
Validation set: 15265 samples (positive: 1704, 11.16%)
Test set: 15265 samples (positive: 1703, 11.16%)
Time taken for splitting data: 0.09 seconds

Applying SMOTE for handling class imbalance...
After SMOTE - Training samples: 126572
Class distribution after SMOTE: [63286 63286]
Time taken for SMOTE: 0.13 seconds

Defining optimized models...

Training XGBoost...
Time taken for training XGBoost: 1.16 seconds

XGBoost Results (threshold = 0.5):
* AUC Score: 0.661
* Precision: 0.153
* Recall: 0.732
* F1-Score: 0.252

Confusion Matrix:
[[6627 6934]
 [ 456 1248]]

Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.49      0.64     13561
           1       0.15      0.73      0.25      1704

    accuracy                           0.52     15265
   macro avg       0.54      0.61      0.45     15265
weighted avg       0.85      0.52      0.60     15265

Time taken for evaluating XGBoost: 0.08 seconds

Finding optimal threshold for XGBoost...
Optimal threshold: 0.60

XGBoost Results (threshold = 0.6000000000000002):
* AUC Score: 0.661
* Precision: 0.196
* Recall: 0.451
* F1-Score: 0.274

Confusion Matrix:
[[10411  3150]
 [  935   769]]

Classification Report:
              precision    recall  f1-score   support
```