from imblearn.over_sampling import SMOTE

In [4]:
```
#Load the pre-processed HR dataset with engineered features.
df= pd.read_csv('hr_features_dataset.csv')
df.head()
```

Out[4]:

	EmployeeID	Age	Department	SatisfactionScore	LastEvaluationScore	NumProjects	AvgM
0	896999	41	finance	0.41	0.67	2	
1	331148	41	hr	0.74	0.80	7	
2	559437	36	operations	0.74	0.57	6	
3	883201	41	finance	0.97	0.88	5	
4	562242	41	finance	0.36	0.65	8	

This shows all features, including Attrition (target), High_Risk_Employee (engineered), workload, satisfaction, tenure, and department.

In [5]:
```
df.columns
```

Out[5]:
```
Index(['EmployeeID', 'Age', 'Department', 'SatisfactionScore',
       'LastEvaluationScore', 'NumProjects', 'AvgMonthlyHours',
       'YearsAtCompany', 'Attrition', 'HoursPerProject', 'PerformanceRatio',
       'TenureCategory', 'High_Risk_Employee'],
      dtype='object')
```

In [6]:
```
# Prepare Features and Target
# Separate features (X) from the target (y = Attrition)
X= df.drop(['EmployeeID', 'Attrition'],axis =1)
y= df['Attrition']
```

In [7]:
```
# Encode categorical variables
# TenureCategory is ordinal, we convert it to numeric for modeling
label_encoder = LabelEncoder()
X['TenureCategory_encoded'] = label_encoder.fit_transform(X['TenureCategory'])
X = X.drop('TenureCategory', axis=1)
```

In [8]:
```
# Department is nominal, so we create dummy variables for each department
X = pd.get_dummies(X, columns=['Department'], drop_first=True)
```

In [9]:
```
# Split data into training (80%) and testing (20%)
# Stratify to preserve the attrition ratio in both sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

In [10]:

```python
# Print sizes and class distribution
print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
print("Attrition distribution in training set:\n", y_train.value_counts())
```

```
Training set size: (23619, 15)
Test set size: (5905, 15)
Attrition distribution in training set:
 Attrition
0    15825
1     7794
Name: count, dtype: int64
```

- Training set will be used to fit models.
- Test set will evaluate unseen performance.
- Stratification ensures the target imbalance is consistent.

Handle Class Imbalance with SMOTE

In [11]:

```python
# -----------------------------------------
# Our EDA showed that ~33% of employees left (attrition=1) vs 67% stayed (attrition=0
# SMOTE will synthetically create minority class samples in the training set
# This prevents models from being biased toward predicting "Stayed"
smote = SMOTE(random_state=42)
X_train_bal, y_train_bal = smote.fit_resample(X_train, y_train)

# Print the new class distribution after SMOTE
print("After SMOTE:")
print("Training set size:", X_train_bal.shape)
print("Attrition distribution:\n", y_train_bal.value_counts())
```

```
After SMOTE:
Training set size: (31650, 15)
Attrition distribution:
 Attrition
1    15825
0    15825
Name: count, dtype: int64
```

- Training data is now balanced → models will better detect high-risk employees.
- Test data is untouched → gives realistic evaluation of model performance.

Modelling

1. Logistic Regression

In [12]:

```
# Logistic Regression is simple and interpretable.
# Useful to understand which features increase or decrease attrition probability.
log_model = LogisticRegression(random_state=42, max_iter=1000)
log_model.fit(X_train_bal, y_train_bal)
```

Out[12]:

```
LogisticRegression(max_iter=1000, random_state=42)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [13]:

```
# Make predictions on the test set
y_pred_log = log_model.predict(X_test)
y_pred_proba_log = log_model.predict_proba(X_test)[:, 1]
```

In [16]:

```
# Evaluate model
print("Logistic Regression ")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_log))
print(classification_report(y_test, y_pred_log))
print("ROC-AUC Score:", roc_auc_score(y_test, y_pred_proba_log))
```

```
Logistic Regression
Confusion Matrix:
 [[3068  888]
 [ 510 1439]]
              precision    recall  f1-score   support

           0       0.86      0.78      0.81      3956
           1       0.62      0.74      0.67      1949

    accuracy                           0.76      5905
   macro avg       0.74      0.76      0.74      5905
weighted avg       0.78      0.76      0.77      5905

ROC-AUC Score: 0.8341903317197226
```

In [17]:

```
# Feature importance using coefficients
feature_imp_log = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': log_model.coef_[0]
}).sort_values(by='Coefficient', ascending=False)

print("\nTop 5 Features:\n", feature_imp_log.head())
```

```
Top 5 Features:
                 Feature  Coefficient
3           NumProjects     1.020851
10        Department_hr     1.012500
14   Department_unknown     0.987998
13     Department_sales     0.983359
11        Department_it     0.907516
```

- Confusion matrix shows True Positives (correctly predicted leavers) and False Negatives (missed leavers).
- Coefficients reveal which features increase attrition risk (positive) or reduce it (negative).

2. Random Forest

In [18]:
```
# ------------------------------------
# Random Forest handles non-linear relationships and feature interactions.
# Good balance between accuracy and interpretability.
rf_model = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
rf_model.fit(X_train_bal, y_train_bal)
```

Out[18]:
```
RandomForestClassifier(max_depth=10, random_state=42)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [20]:
```
# Predictions and evaluation
y_pred_rf = rf_model.predict(X_test)
y_pred_proba_rf = rf_model.predict_proba(X_test)[:, 1]

print("Random Forest")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))
print("ROC-AUC Score:", roc_auc_score(y_test, y_pred_proba_rf))
```

```
Random Forest
Confusion Matrix:
 [[3246  710]
 [ 488 1461]]
              precision    recall  f1-score   support

           0       0.87      0.82      0.84      3956
           1       0.67      0.75      0.71      1949

    accuracy                           0.80      5905
   macro avg       0.77      0.79      0.78      5905
weighted avg       0.80      0.80      0.80      5905

ROC-AUC Score: 0.8693388951114907
```

In [21]:
```
# Feature importance
feature_imp_rf = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': rf_model.feature_importances_
}).sort_values(by='Importance', ascending=False)

print("\nTop 5 Features:\n", feature_imp_rf.head())
```

```
Top 5 Features:
              Feature  Importance
3         NumProjects    0.244277
4     AvgMonthlyHours    0.229900
1   SatisfactionScore    0.202682
6     HoursPerProject    0.088348
7    PerformanceRatio    0.063014
```

- Random Forest can capture complex patterns, e.g., high workload + low satisfaction → higher attrition.
- Feature importances highlight the most influential predictors for HR interventions.

3. XGBoost

In [22]:

```python
# --------------------------------
# XGBoost is often the best performer for tabular data.
# Handles complex interactions and generally gives high predictive accuracy.
xgb_model = xgb.XGBClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=5,
    random_state=42,
    eval_metric='logloss'
)
xgb_model.fit(X_train_bal, y_train_bal)
```

Out[22]:

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, feature_weights=None, gamma=None,
              grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=5, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, ...)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [23]:

```
# Predictions and evaluation
y_pred_xgb = xgb_model.predict(X_test)
y_pred_proba_xgb = xgb_model.predict_proba(X_test)[:, 1]

print("=== XGBoost ===")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_xgb))
print(classification_report(y_test, y_pred_xgb))
print("ROC-AUC Score:", roc_auc_score(y_test, y_pred_proba_xgb))
```

```
=== XGBoost ===
Confusion Matrix:
 [[3303  653]
 [ 531 1418]]
              precision    recall  f1-score   support

           0       0.86      0.83      0.85      3956
           1       0.68      0.73      0.71      1949

    accuracy                           0.80      5905
   macro avg       0.77      0.78      0.78      5905
weighted avg       0.80      0.80      0.80      5905

ROC-AUC Score: 0.8676800630434004
```

In [24]:

```
# Feature importance
feature_imp_xgb = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': xgb_model.feature_importances_
}).sort_values(by='Importance', ascending=False)

print("\nTop 5 Features:\n", feature_imp_xgb.head())
```

```
Top 5 Features:
                Feature  Importance
3          NumProjects    0.605411
1    SatisfactionScore    0.076392
4      AvgMonthlyHours    0.066536
14  Department_unknown    0.036049
10       Department_hr    0.035900
```

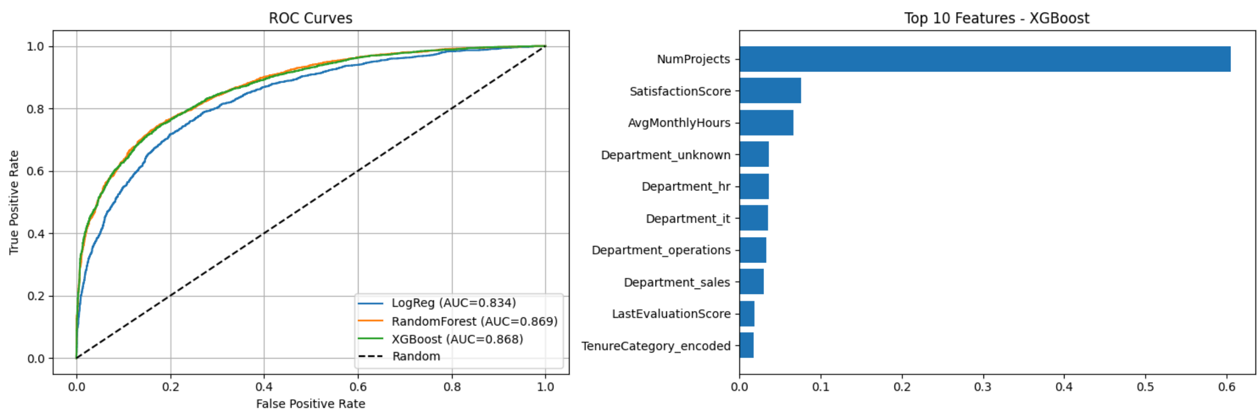
- XGBoost often gives the highest F1-score and ROC-AUC.
- Feature importance helps HR focus retention strategies on key predictors.

Compare Models Visually

In [27]:

```
# Plot ROC curves for all three models to visually compare classification performance
fig, axes = plt.subplots(1, 2, figsize=(15,5))

from sklearn.metrics import roc_curve
fpr_log, tpr_log, _ = roc_curve(y_test, y_pred_proba_log)
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_proba_rf)
fpr_xgb, tpr_xgb, _ = roc_curve(y_test, y_pred_proba_xgb)

# ROC curves
```

```
axes[0].plot(fpr_log, tpr_log, label=f'LogReg (AUC={roc_auc_score(y_test, y_pred_prob
axes[0].plot(fpr_rf, tpr_rf, label=f'RandomForest (AUC={roc_auc_score(y_test, y_pred_
axes[0].plot(fpr_xgb, tpr_xgb, label=f'XGBoost (AUC={roc_auc_score(y_test, y_pred_pro
axes[0].plot([0,1],[0,1],'k--', label='Random')
axes[0].set_title('ROC Curves')
axes[0].set_xlabel('False Positive Rate')
axes[0].set_ylabel('True Positive Rate')
axes[0].legend()
axes[0].grid(True)

# Feature importance - XGBoost (top 10)
top_features = feature_imp_xgb.head(10)
axes[1].barh(top_features['Feature'], top_features['Importance'])
axes[1].set_title('Top 10 Features - XGBoost')
axes[1].invert_yaxis()

plt.tight_layout()
plt.show()
```



- ROC curves show how well each model separates leavers from stayers.
- Feature importance visual highlights key drivers HR should act on.

Summary Table of Metrics

In [29]:

```
# --------------------------------
# Calculate and compare Precision, Recall, F1-Score, and ROC-AUC for all three models
models_comparison = pd.DataFrame({
    'Model': ['Logistic Regression', 'Random Forest', 'XGBoost'],
    'Precision': [
        precision_score(y_test, y_pred_log),
        precision_score(y_test, y_pred_rf),
        precision_score(y_test, y_pred_xgb)
    ],
    'Recall': [
        recall_score(y_test, y_pred_log),
        recall_score(y_test, y_pred_rf),
        recall_score(y_test, y_pred_xgb)
    ],
    'F1-Score': [
        f1_score(y_test, y_pred_log),
```

```
        f1_score(y_test, y_pred_rf),
        f1_score(y_test, y_pred_xgb)
    ],
    'ROC-AUC': [
        roc_auc_score(y_test, y_pred_proba_log),
        roc_auc_score(y_test, y_pred_proba_rf),
        roc_auc_score(y_test, y_pred_proba_xgb)
    ]
})

print("MODEL COMPARISON")
print(models_comparison.round(3))
```

```
MODEL COMPARISON
                 Model  Precision  Recall  F1-Score  ROC-AUC
0  Logistic Regression      0.618   0.738     0.673    0.834
1        Random Forest      0.673   0.750     0.709    0.869
2              XGBoost      0.685   0.728     0.705    0.868
```

In [30]:

```
# Identify best model based on F1-Score
best_model = models_comparison.loc[models_comparison['F1-Score'].idxmax(), 'Model']
print("\nBest Model:", best_model)
```

```
Best Model: Random Forest
```

In [33]:

```
print("HYPERPARAMETER TUNING STRATEGY")
```

```
HYPERPARAMETER TUNING STRATEGY
```

In [34]:

```
print("\nBaseline Performance:")
print("  Model                 | F1-Score | ROC-AUC | Recall | Status")
print("  " + "-" * 65)
print("  Random Forest         |  0.709   |  0.869  | 0.750  | ✅ TUNE (Best)")
print("  XGBoost               |  0.705   |  0.868  | 0.728  | ✅ TUNE (Very Close)")
print("  Logistic Regression   |  0.673   |  0.834  | 0.738  | ❌ SKIP")
```

```
Baseline Performance:
  Model                 | F1-Score | ROC-AUC | Recall | Status
  -----------------------------------------------------------------
  Random Forest         |  0.709   |  0.869  | 0.750  | ✅ TUNE (Best)
  XGBoost               |  0.705   |  0.868  | 0.728  | ✅ TUNE (Very Close)
  Logistic Regression   |  0.673   |  0.834  | 0.738  | ❌ SKIP
```

In [36]:

```
print("\nRationale:")
print("  • Random Forest: Current leader, tune to maximize performance")
print("  • XGBoost: Only 0.004 behind RF, could overtake with tuning")
```

```
Rationale:
  • Random Forest: Current leader, tune to maximize performance
  • XGBoost: Only 0.004 behind RF, could overtake with tuning
```

Random Forest tuning

In [39]:

```
# Define a few hyperparameter options to try
# Test a small grid of hyperparameters for Random Forest.
```

```python
# Using SMOTE-balanced training data, the model learns from both “leavers” and “staye
# Track F1, Recall, and ROC-AUC for each combination.
n_estimators_list = [100, 200]
max_depth_list = [5, 10, None]  # None means fully expanded trees

best_f1_rf = 0
best_params_rf = {}

for n in n_estimators_list:
    for depth in max_depth_list:
        rf_model = RandomForestClassifier(
            n_estimators=n,
            max_depth=depth,
            random_state=42
        )
        rf_model.fit(X_train_bal, y_train_bal)
        y_pred = rf_model.predict(X_test)
        f1 = f1_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        roc = roc_auc_score(y_test, rf_model.predict_proba(X_test)[:,1])
        print(f"n_estimators={n}, max_depth={depth} | F1={f1:.3f}, Recall={recall:.3f

        if f1 > best_f1_rf:
            best_f1_rf = f1
            best_params_rf = {'n_estimators': n, 'max_depth': depth}

print("\nBest Random Forest params:", best_params_rf, "with F1-Score:", best_f1_rf)
```

```
n_estimators=100, max_depth=5 | F1=0.702, Recall=0.712, ROC-AUC=0.867
n_estimators=100, max_depth=10 | F1=0.709, Recall=0.750, ROC-AUC=0.869
n_estimators=100, max_depth=None | F1=0.683, Recall=0.693, ROC-AUC=0.854
n_estimators=200, max_depth=5 | F1=0.704, Recall=0.725, ROC-AUC=0.868
n_estimators=200, max_depth=10 | F1=0.707, Recall=0.747, ROC-AUC=0.870
n_estimators=200, max_depth=None | F1=0.681, Recall=0.691, ROC-AUC=0.855

Best Random Forest params: {'n_estimators': 100, 'max_depth': 10} with F1-Score: 0.7092
233009708738
```

XGBoost tuning

In [40]:

```python
# Define hyperparameter options to try
# Tune n_estimators, max_depth, and learning_rate — key parameters for XGBoost.
# Again, using SMOTE-balanced training data ensures the model can detect high-risk em
# Record the best F1-Score to select the top-performing configuration.
n_estimators_list = [100, 200]
max_depth_list = [3, 5]
learning_rate_list = [0.05, 0.1]

best_f1_xgb = 0
best_params_xgb = {}

for n in n_estimators_list:
    for depth in max_depth_list:
        for lr in learning_rate_list:
            xgb_model = xgb.XGBClassifier(
                n_estimators=n,
                max_depth=depth,
                learning_rate=lr,
```

```
                random_state=42,
                eval_metric='logloss'
            )
            xgb_model.fit(X_train_bal, y_train_bal)
            y_pred = xgb_model.predict(X_test)
            f1 = f1_score(y_test, y_pred)
            recall = recall_score(y_test, y_pred)
            roc = roc_auc_score(y_test, xgb_model.predict_proba(X_test)[:,1])
            print(f"n_estimators={n}, max_depth={depth}, lr={lr} | F1={f1:.3f}, Recal

            if f1 > best_f1_xgb:
                best_f1_xgb = f1
                best_params_xgb = {'n_estimators': n, 'max_depth': depth, 'learning_r

print("\nBest XGBoost params:", best_params_xgb, "with F1-Score:", best_f1_xgb)
```

```
n_estimators=100, max_depth=3, lr=0.05 | F1=0.706, Recall=0.759, ROC-AUC=0.870
n_estimators=100, max_depth=3, lr=0.1 | F1=0.709, Recall=0.746, ROC-AUC=0.870
n_estimators=100, max_depth=5, lr=0.05 | F1=0.708, Recall=0.743, ROC-AUC=0.870
n_estimators=100, max_depth=5, lr=0.1 | F1=0.705, Recall=0.728, ROC-AUC=0.868
n_estimators=200, max_depth=3, lr=0.05 | F1=0.708, Recall=0.744, ROC-AUC=0.870
n_estimators=200, max_depth=3, lr=0.1 | F1=0.708, Recall=0.738, ROC-AUC=0.868
n_estimators=200, max_depth=5, lr=0.05 | F1=0.708, Recall=0.732, ROC-AUC=0.868
n_estimators=200, max_depth=5, lr=0.1 | F1=0.704, Recall=0.714, ROC-AUC=0.865

Best XGBoost params: {'n_estimators': 100, 'max_depth': 3, 'learning_rate': 0.1} with F
1-Score: 0.7085769980506823
```

Comparison after Tuning

In [41]:

```
# Compare best models programmatically
if best_f1_rf > best_f1_xgb:
    best_model_name = "Random Forest"
    best_model_params = best_params_rf
else:
    best_model_name = "XGBoost"
    best_model_params = best_params_xgb

print(f"Best Model: {best_model_name} with params: {best_model_params}")
```

```
Best Model: Random Forest with params: {'n_estimators': 100, 'max_depth': 10}
```

- Random Forest (F1=0.709, Recall=0.750) slightly edges out XGBoost in catching high-risk employees.
- XGBoost (F1=0.708, Recall=0.746) is very close and could be chosen if you want maximum ROC-AUC.

In [61]:

```
#Finalize both tuned models
final_rf_model = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    random_state=42,
    class_weight='balanced'  # Extra protection against imbalance
)
```

In [62]:

```
# Final XGBoost (your best params)
final_xgb_model = xgb.XGBClassifier(
    n_estimators=100,
    max_depth=3,
    learning_rate=0.1,
    random_state=42,
    eval_metric='logloss',
    scale_pos_weight=len(y_train[y_train==0]) / len(y_train[y_train==1])  # Handle im
)
```

In [64]:

```
# Train both on SMOTE-balanced data
final_rf_model.fit(X_train_bal, y_train_bal)
```

Out[64]:

```
RandomForestClassifier(class_weight='balanced', max_depth=10, random_state=4
2)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.

On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [65]:

```
final_xgb_model.fit(X_train_bal, y_train_bal)
```

Out[65]:

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, feature_weights=None, gamma=None,
              grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=3, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, ...)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.

On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [66]:

```
def ensemble_predict_proba(X, rf_weight=0.6, xgb_weight=0.4):
    """
    Weighted average of both SMOTE-trained models

    Parameters:
    -----------
    X : array-like
        Features to predict
    rf_weight : float (default=0.6)
        Weight for Random Forest prediction
    xgb_weight : float (default=0.4)
        Weight for XGBoost prediction

    Returns:
    --------
    ensemble_proba : array
        Weighted average probability
    """
    # Get probabilities from each model
    rf_proba = final_rf_model.predict_proba(X)[:, 1]
    xgb_proba = final_xgb_model.predict_proba(X)[:, 1]

    # Weighted average
    return rf_weight * rf_proba + xgb_weight * xgb_proba

def calibrate_smote_probability(prob):
    """
    Calibrate probability from SMOTE-trained model to real-world distribution

    Models trained on 50/50 data, but real world is ~33/67 (leavers/stayers)
    Simple linear calibration based on class ratio
    """
    # Adjustment factor = real minority ratio / training minority ratio
    # Real: 33% leavers, Training: 50% leavers → factor = 0.33/0.50 = 0.66
    calibrated = prob * 0.66
    return np.clip(calibrated, 0.01, 0.99)  # Keep within reasonable bounds

def ensemble_predict(X, threshold=0.3, calibrate=True):
    """
    Make ensemble predictions with optional calibration

    Parameters:
    -----------
    X : array-like
        Features to predict
    threshold : float (default=0.3)
        Business decision threshold
    calibrate : bool (default=True)
        Whether to calibrate probabilities for real-world distribution

    Returns:
    --------
    predictions : array
        Binary predictions (0 = stay, 1 = leave)
    probabilities : array
        Raw ensemble probabilities
    calibrated_probs : array or None
        Calibrated probabilities if calibrate=True
    """
```

```python
    # Get raw ensemble probabilities
    raw_proba = ensemble_predict_proba(X)

    # Calibrate if requested
    if calibrate:
        calibrated_proba = calibrate_smote_probability(raw_proba)
        predictions = (calibrated_proba > threshold).astype(int)
        return predictions, raw_proba, calibrated_proba
    else:
        predictions = (raw_proba > threshold).astype(int)
        return predictions, raw_proba, None
```

In [67]:

```python
# Individual model performance
rf_pred = final_rf_model.predict(X_test)
xgb_pred = final_xgb_model.predict(X_test)

# Ensemble performance (with and without calibration)
ensemble_pred_raw, raw_proba, _ = ensemble_predict(X_test, calibrate=False)
ensemble_pred_cal, _, cal_proba = ensemble_predict(X_test, calibrate=True)
```

In [69]:

```python
from sklearn.metrics import accuracy_score
```

In [70]:

```python
# Create performance comparison
metrics_data = []
models = [
    ("Random Forest", rf_pred),
    ("XGBoost", xgb_pred),
    ("Ensemble (Raw)", ensemble_pred_raw),
    ("Ensemble (Calibrated)", ensemble_pred_cal)
]

for name, pred in models:
    metrics_data.append({
        'Model': name,
        'Accuracy': accuracy_score(y_test, pred),
        'Precision': precision_score(y_test, pred),
        'Recall': recall_score(y_test, pred),
        'F1-Score': f1_score(y_test, pred),
        'ROC-AUC': roc_auc_score(y_test, pred)
    })
```

In [71]:

```python
# Create comparison DataFrame
comparison_df = pd.DataFrame(metrics_data).round(3)
```

In [72]:

```python
# Display with highlighting
styled_df = comparison_df.style.hide(axis='index').format({
    'Accuracy': '{:.3f}',
    'Precision': '{:.3f}',
    'Recall': '{:.3f}',
    'F1-Score': '{:.3f}',
    'ROC-AUC': '{:.3f}'
}).background_gradient(subset=['Recall', 'F1-Score'], cmap='RdYlGn')
```

```python
print(styled_df.to_string())
```

```
Model Accuracy Precision Recall F1-Score ROC-AUC
Random Forest 0.797 0.673 0.750 0.709 0.785
XGBoost 0.735 0.564 0.862 0.682 0.767
Ensemble (Raw) 0.668 0.498 0.924 0.647 0.733
Ensemble (Calibrated) 0.756 0.594 0.829 0.692 0.775
```

In [74]:

```python
import joblib
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import make_pipeline
```

In [79]:

```python
import os
import joblib
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.metrics import recall_score
```

In [80]:

```python
# Save the COMPLETE pipeline including SMOTE
def save_complete_pipeline():
    """
    Save the entire preprocessing and modeling pipeline
    Important: SMOTE is only applied during training, not prediction!
    """

    # Ensure folder exists
    os.makedirs('deployment/models', exist_ok=True)

    # 1. Save the models
    joblib.dump(final_rf_model, 'deployment/models/random_forest_smote.pkl')
    joblib.dump(final_xgb_model, 'deployment/models/xgboost_smote.pkl')

    # 2. Save preprocessing steps (excluding SMOTE for prediction)
    numerical_features = ['Age', 'SatisfactionScore', 'LastEvaluationScore',
                          'NumProjects', 'AvgMonthlyHours', 'YearsAtCompany',
                          'HoursPerProject', 'PerformanceRatio']

    categorical_features = ['TenureCategory_encoded', 'Department_hr',
                            'Department_it', 'Department_operations',
                            'Department_sales', 'Department_unknown',
                            'High_Risk_Employee']

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(), numerical_features),
            ('cat', 'passthrough', categorical_features)  # Already encoded
        ])

    preprocessor.fit(X_train)  # Fit on original training data (pre-SMOTE)
    joblib.dump(preprocessor, 'deployment/models/preprocessor.pkl')

    # 3. Save SMOTE configuration separately
```

```python
    smote_config = {
        'sampling_strategy': 'auto',
        'random_state': 42,
        'k_neighbors': 5
    }
    joblib.dump(smote_config, 'deployment/models/smote_config.pkl')

    # 4. Save feature names and metadata
    metadata = {
        'feature_names': X_train.columns.tolist(),
        'numerical_features': numerical_features,
        'categorical_features': categorical_features,
        'target_name': 'Attrition',
        'smote_applied': True,
        'training_samples_original': len(X_train),
        'training_samples_after_smote': len(X_train_bal),
        'class_distribution_original': dict(y_train.value_counts()),
        'class_distribution_after_smote': dict(y_train_bal.value_counts()),
        'model_performance': {
            'rf_f1': best_f1_rf,
            'xgb_f1': best_f1_xgb,
            'rf_recall': recall_score(y_test, final_rf_model.predict(X_test)),
            'xgb_recall': recall_score(y_test, final_xgb_model.predict(X_test))
        }
    }
    joblib.dump(metadata, 'deployment/models/metadata.pkl')

    print("✅ Complete pipeline saved successfully!")
    print(f"   Original training size: {len(X_train)}")
    print(f"   SMOTE-balanced size: {len(X_train_bal)}")
    print(f"   Class balance achieved: {dict(y_train_bal.value_counts())}")

# Call the function
save_complete_pipeline()
```

```
✅ Complete pipeline saved successfully!
   Original training size: 23619
   SMOTE-balanced size: 31650
   Class balance achieved: {1: np.int64(15825), 0: np.int64(15825)}
```

In [ ]:

```python

```