# Advanced Modeling Techniques with Amazon SageMaker

This Jupyter notebook guides you through implementing advanced modeling techniques using Amazon SageMaker, focusing on ensemble methods, model evaluation, and optimization techniques as covered in Chapter 5.

## Prerequisites

* An AWS account with SageMaker access
* Basic understanding of Python and machine learning concepts
* Familiarity with Jupyter notebooks

## Lab Overview

In this lab, you will:

1. Set up a SageMaker environment
2. Prepare data for modeling
3. Implement ensemble methods (bagging, boosting, and stacking)
4. Create a comprehensive model evaluation framework
5. Apply advanced optimization techniques
6. Clean up all resources

Let's get started!

## 1. Environment Setup

First, let's set up our SageMaker environment and install the necessary libraries:

# Install required packages  
!pip3 install -q pandas numpy matplotlib seaborn scikit-learn xgboost

# Import necessary libraries  
import boto3  
import sagemaker  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.datasets import load\_breast\_cancer  
from sklearn.model\_selection import train\_test\_split, cross\_val\_score, KFold, StratifiedKFold  
from sklearn.metrics import (accuracy\_score, precision\_score, recall\_score, f1\_score,   
 confusion\_matrix, roc\_curve, auc, precision\_recall\_curve)  
from sklearn.preprocessing import StandardScaler  
from sklearn.ensemble import RandomForestClassifier, StackingClassifier  
from sklearn.linear\_model import LogisticRegression  
from sklearn.svm import SVC  
import xgboost as xgb  
  
# Set up the SageMaker session  
session = sagemaker.Session()  
role = sagemaker.get\_execution\_role()  
region = boto3.Session().region\_name  
  
print(f"SageMaker session established in region: {region}")  
print(f"Using role: {role}")

/opt/conda/lib/python3.11/site-packages/pydantic/\_internal/\_fields.py:192: UserWarning: Field name "json" in "MonitoringDatasetFormat" shadows an attribute in parent "Base"  
 warnings.warn(

sagemaker.config INFO - Not applying SDK defaults from location: /etc/xdg/sagemaker/config.yaml  
sagemaker.config INFO - Not applying SDK defaults from location: /home/sagemaker-user/.config/sagemaker/config.yaml  
SageMaker session established in region: us-east-1  
Using role: arn:aws:iam::495304082646:role/service-role/AmazonSageMaker-ExecutionRole-20250308T154241

## 2. Data Preparation

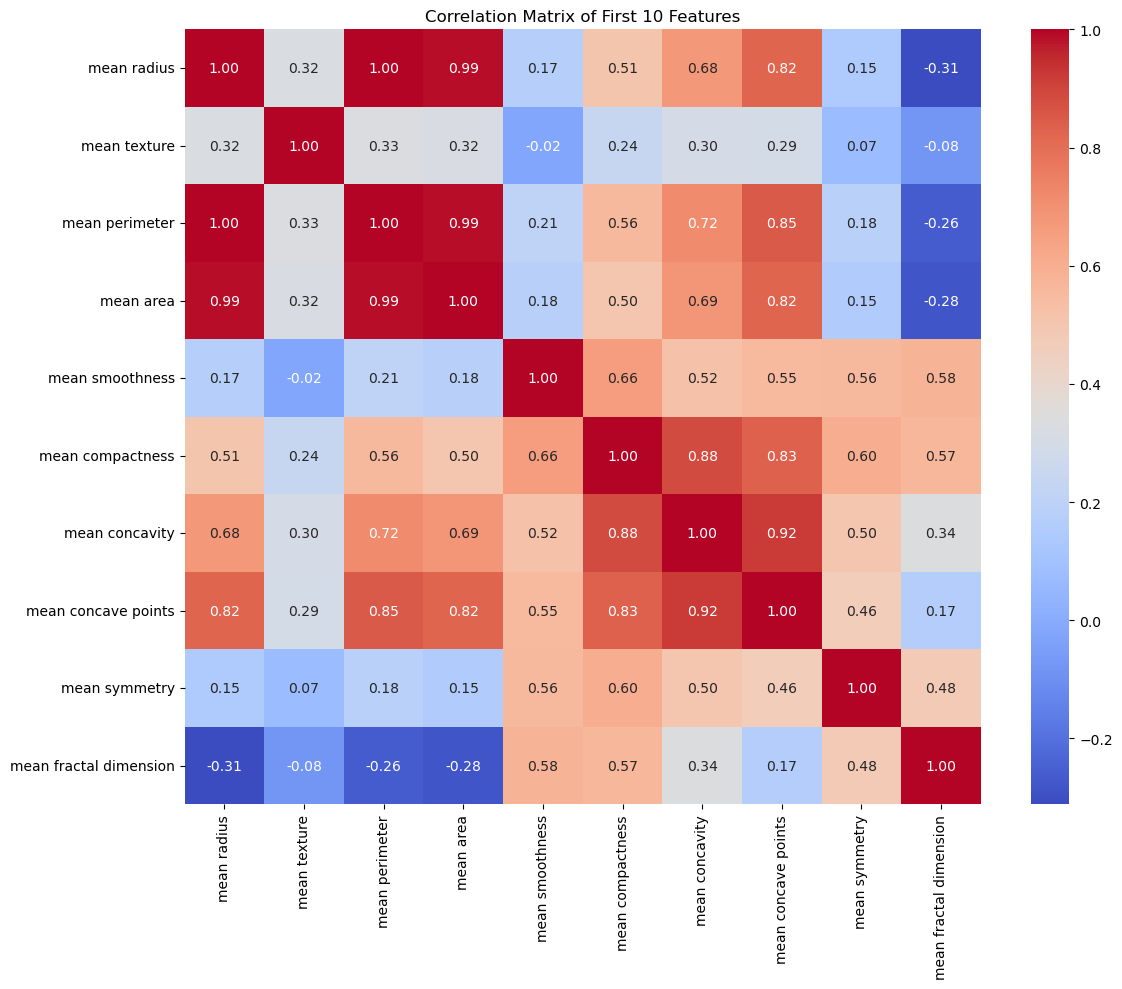
We'll use the Breast Cancer Wisconsin dataset for this lab, which is a binary classification problem:

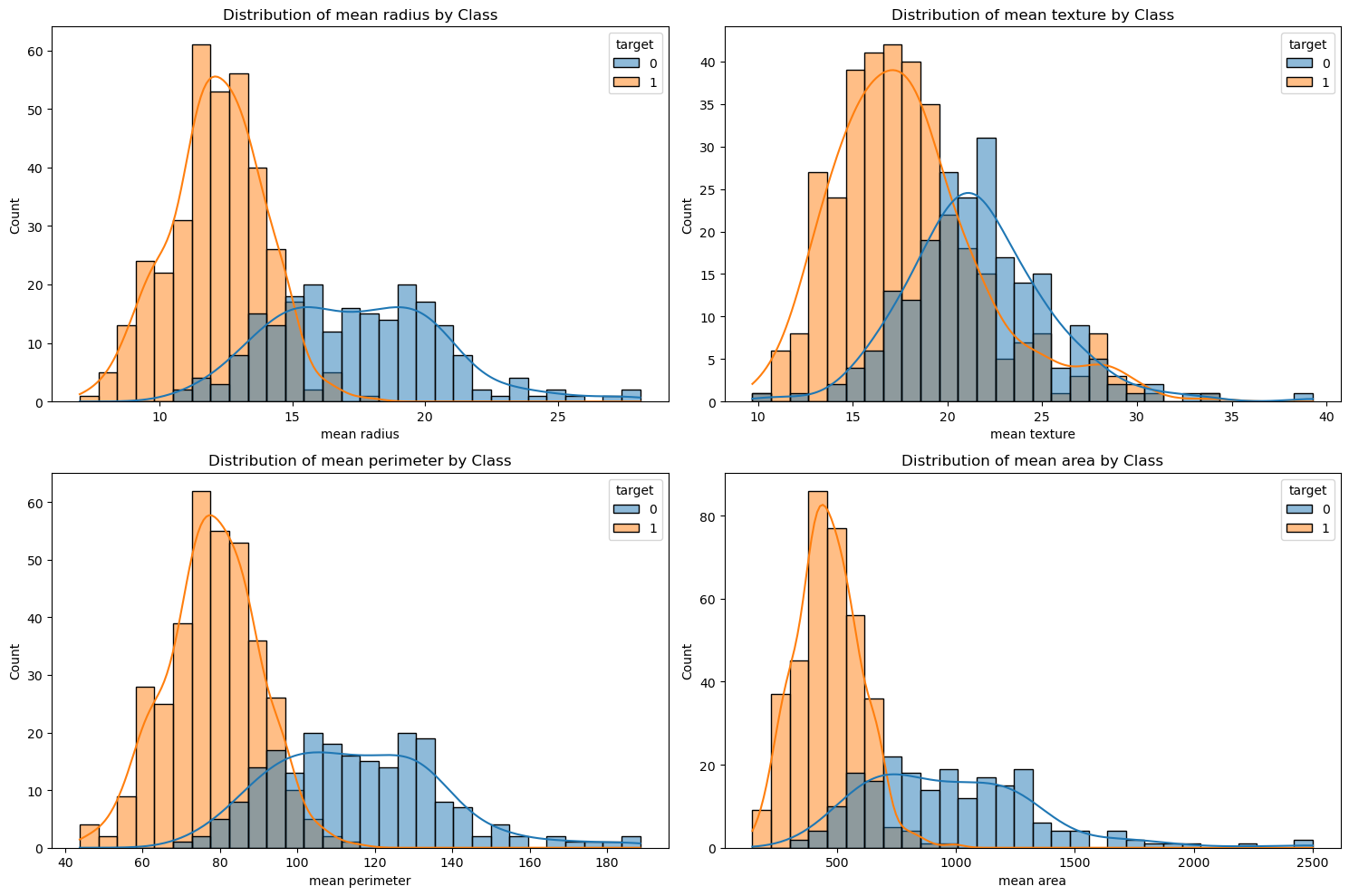
# Load the dataset  
cancer = load\_breast\_cancer()  
X, y = cancer.data, cancer.target  
  
# Print dataset information  
print(f"Dataset shape: {X.shape}")  
print(f"Number of features: {len(cancer.feature\_names)}")  
print(f"Target distribution: {np.bincount(y)}")  
print(f"Class names: {cancer.target\_names}")  
  
# Split the data into training and testing sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# Scale the features  
scaler = StandardScaler()  
X\_train\_scaled = scaler.fit\_transform(X\_train)  
X\_test\_scaled = scaler.transform(X\_test)  
  
print(f"Training set shape: {X\_train.shape}")  
print(f"Testing set shape: {X\_test.shape}")

Dataset shape: (569, 30)  
Number of features: 30  
Target distribution: [212 357]  
Class names: ['malignant' 'benign']  
Training set shape: (455, 30)  
Testing set shape: (114, 30)

Let's visualize some of the data to understand it better:

# Create a DataFrame for easier visualization  
cancer\_df = pd.DataFrame(X, columns=cancer.feature\_names)  
cancer\_df['target'] = y  
  
# Plot correlation matrix for the first 10 features  
plt.figure(figsize=(12, 10))  
correlation\_matrix = cancer\_df.iloc[:, :10].corr()  
sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', fmt='.2f')  
plt.title('Correlation Matrix of First 10 Features')  
plt.tight\_layout()  
plt.show()  
  
# Plot distribution of a few key features by class  
plt.figure(figsize=(15, 10))  
features\_to\_plot = ['mean radius', 'mean texture', 'mean perimeter', 'mean area']  
for i, feature in enumerate(features\_to\_plot):  
 plt.subplot(2, 2, i+1)  
 sns.histplot(data=cancer\_df, x=feature, hue='target', kde=True, bins=30)  
 plt.title(f'Distribution of {feature} by Class')  
plt.tight\_layout()  
plt.show()



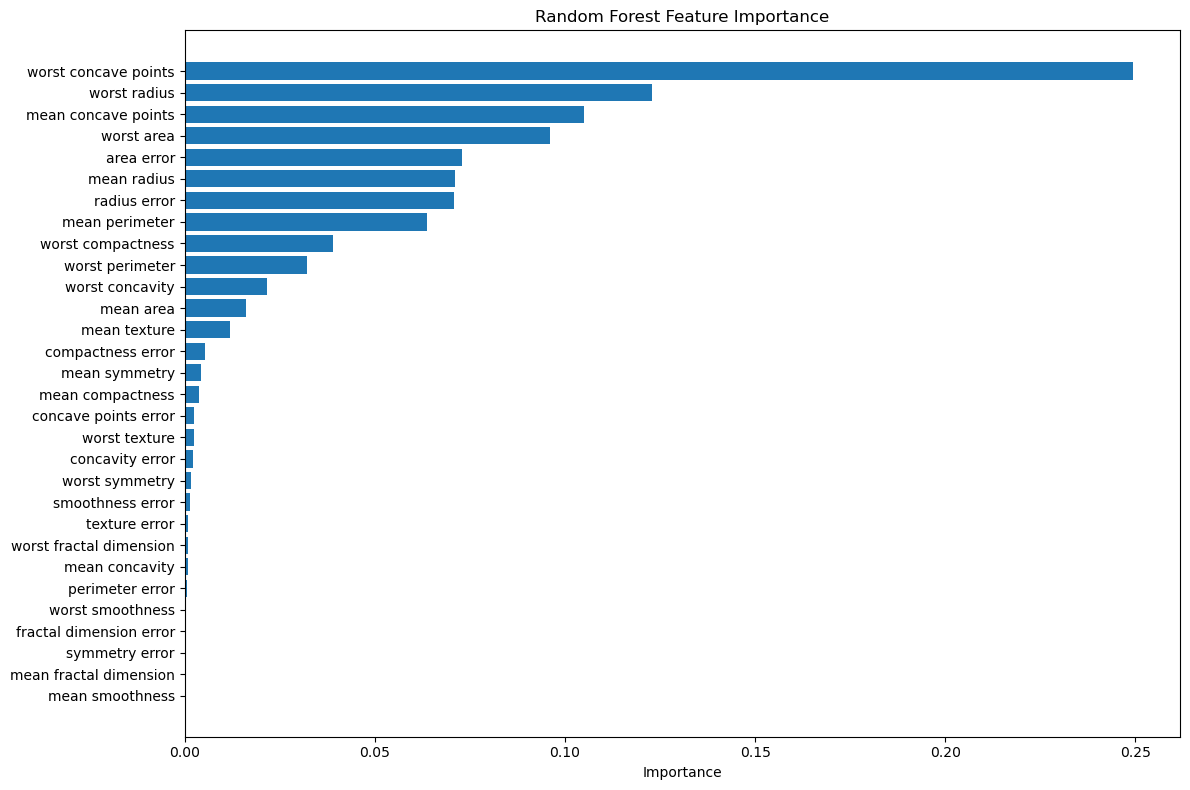


## 3. Implementing Ensemble Methods

### 3.1 Bagging with Random Forest

# Train a Random Forest classifier (bagging ensemble)  
rf\_model = RandomForestClassifier(n\_estimators=10, max\_depth=3, random\_state=42)  
rf\_model.fit(X\_train\_scaled, y\_train)  
  
# Make predictions  
rf\_preds = rf\_model.predict(X\_test\_scaled)  
rf\_probs = rf\_model.predict\_proba(X\_test\_scaled)[:, 1]  
  
print("Random Forest (Bagging) Performance:")  
print(f"Accuracy: {accuracy\_score(y\_test, rf\_preds):.4f}")  
print(f"Precision: {precision\_score(y\_test, rf\_preds):.4f}")  
print(f"Recall: {recall\_score(y\_test, rf\_preds):.4f}")  
print(f"F1 Score: {f1\_score(y\_test, rf\_preds):.4f}")  
  
# Plot feature importance  
plt.figure(figsize=(12, 8))  
feature\_importance = rf\_model.feature\_importances\_  
sorted\_idx = np.argsort(feature\_importance)  
plt.barh(range(len(sorted\_idx)), feature\_importance[sorted\_idx], align='center')  
plt.yticks(range(len(sorted\_idx)), [cancer.feature\_names[i] for i in sorted\_idx])  
plt.title('Random Forest Feature Importance')  
plt.xlabel('Importance')  
plt.tight\_layout()  
plt.show()

Random Forest (Bagging) Performance:  
Accuracy: 0.9561  
Precision: 0.9583  
Recall: 0.9718  
F1 Score: 0.9650

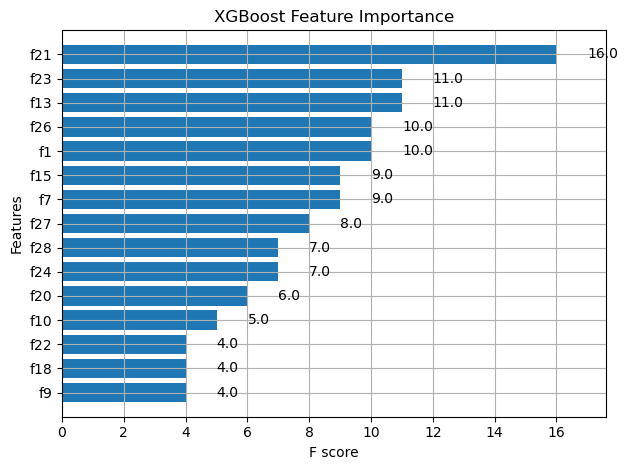


### 3.2 Boosting with XGBoost

# Train an XGBoost model (boosting ensemble)  
dtrain = xgb.DMatrix(X\_train\_scaled, label=y\_train)  
dtest = xgb.DMatrix(X\_test\_scaled, label=y\_test)  
  
params = {  
 'objective': 'binary:logistic',  
 'eval\_metric': 'logloss',  
 'max\_depth': 3,  
 'eta': 0.3,  
 'subsample': 0.8,  
 'colsample\_bytree': 0.8  
}  
  
xgb\_model = xgb.train(  
 params,   
 dtrain,   
 num\_boost\_round=100,  
 evals=[(dtest, 'test')],  
 early\_stopping\_rounds=10,  
 verbose\_eval=False  
)  
  
# Make predictions  
xgb\_probs = xgb\_model.predict(dtest)  
xgb\_preds = [1 if prob > 0.5 else 0 for prob in xgb\_probs]  
  
print("\nXGBoost (Boosting) Performance:")  
print(f"Accuracy: {accuracy\_score(y\_test, xgb\_preds):.4f}")  
print(f"Precision: {precision\_score(y\_test, xgb\_preds):.4f}")  
print(f"Recall: {recall\_score(y\_test, xgb\_preds):.4f}")  
print(f"F1 Score: {f1\_score(y\_test, xgb\_preds):.4f}")  
  
# Plot feature importance  
plt.figure(figsize=(12, 8))  
xgb.plot\_importance(xgb\_model, max\_num\_features=15, height=0.8)  
plt.title('XGBoost Feature Importance')  
plt.tight\_layout()  
plt.show()

XGBoost (Boosting) Performance:  
Accuracy: 0.9737  
Precision: 0.9722  
Recall: 0.9859  
F1 Score: 0.9790

<Figure size 1200x800 with 0 Axes>



### 3.3 Stacking Ensemble

# Define base models for stacking  
base\_models = [  
 ('rf', RandomForestClassifier(n\_estimators=5, random\_state=42)),  
 ('svm', SVC(probability=True, random\_state=42))  
]  
  
# Define meta-learner  
meta\_learner = LogisticRegression(random\_state=42)  
  
# Create and train stacking ensemble  
stacking\_model = StackingClassifier(  
 estimators=base\_models,  
 final\_estimator=meta\_learner,  
 cv=5  
)  
  
stacking\_model.fit(X\_train\_scaled, y\_train)  
  
# Make predictions  
stacking\_preds = stacking\_model.predict(X\_test\_scaled)  
stacking\_probs = stacking\_model.predict\_proba(X\_test\_scaled)[:, 1]  
  
print("\nStacking Ensemble Performance:")  
print(f"Accuracy: {accuracy\_score(y\_test, stacking\_preds):.4f}")  
print(f"Precision: {precision\_score(y\_test, stacking\_preds):.4f}")  
print(f"Recall: {recall\_score(y\_test, stacking\_preds):.4f}")  
print(f"F1 Score: {f1\_score(y\_test, stacking\_preds):.4f}")

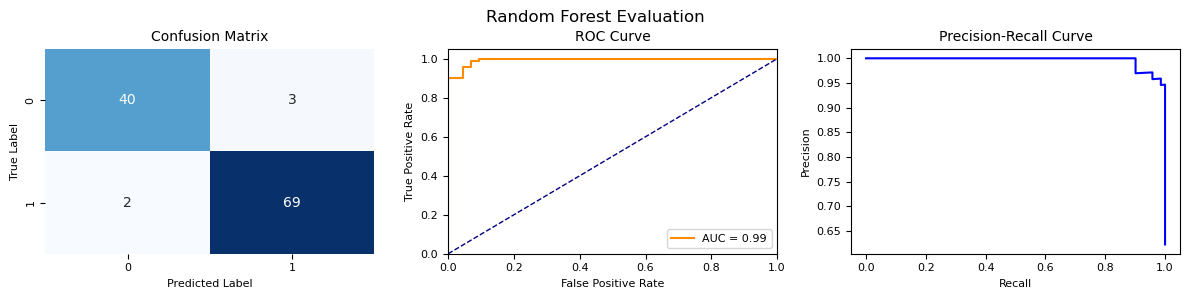
Stacking Ensemble Performance:  
Accuracy: 0.9737  
Precision: 0.9722  
Recall: 0.9859  
F1 Score: 0.9790

## 4. Model Evaluation Framework

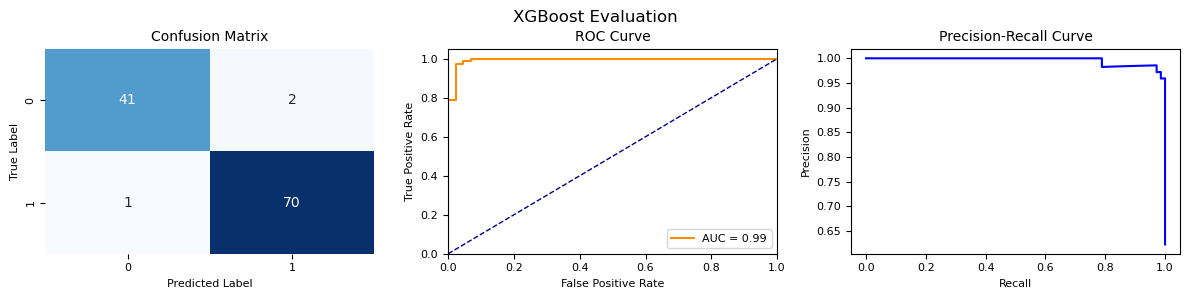
Let's create a comprehensive evaluation framework to compare our models:

def evaluate\_model(y\_true, y\_pred, y\_prob, model\_name, metrics\_dict=None):  
 """  
 Compact model evaluation function for reports  
 """  
 # Calculate metrics  
 metrics = {  
 'accuracy': accuracy\_score(y\_true, y\_pred),  
 'precision': precision\_score(y\_true, y\_pred),  
 'recall': recall\_score(y\_true, y\_pred),  
 'f1': f1\_score(y\_true, y\_pred)  
 }  
   
 # Store metrics if dictionary provided  
 if metrics\_dict is not None:  
 metrics\_dict[model\_name] = metrics  
   
 # Print metrics in compact format  
 print(f"{model\_name}: Acc={metrics['accuracy']:.4f}, Prec={metrics['precision']:.4f}, "  
 f"Rec={metrics['recall']:.4f}, F1={metrics['f1']:.4f}")  
   
 # Create subplot figure for all visualizations  
 fig, ax = plt.subplots(1, 3, figsize=(12, 3))  
   
 # Confusion Matrix  
 cm = confusion\_matrix(y\_true, y\_pred)  
 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax[0], cbar=False)  
 ax[0].set\_title(f'Confusion Matrix', fontsize=10)  
 ax[0].set\_ylabel('True Label', fontsize=8)  
 ax[0].set\_xlabel('Predicted Label', fontsize=8)  
 ax[0].tick\_params(labelsize=8)  
   
 # ROC Curve  
 fpr, tpr, \_ = roc\_curve(y\_true, y\_prob)  
 roc\_auc = auc(fpr, tpr)  
 ax[1].plot(fpr, tpr, color='darkorange', lw=1.5, label=f'AUC = {roc\_auc:.2f}')  
 ax[1].plot([0, 1], [0, 1], color='navy', lw=1, linestyle='--')  
 ax[1].set\_xlim([0.0, 1.0])  
 ax[1].set\_ylim([0.0, 1.05])  
 ax[1].set\_xlabel('False Positive Rate', fontsize=8)  
 ax[1].set\_ylabel('True Positive Rate', fontsize=8)  
 ax[1].set\_title(f'ROC Curve', fontsize=10)  
 ax[1].legend(loc="lower right", fontsize=8)  
 ax[1].tick\_params(labelsize=8)  
   
 # Precision-Recall Curve  
 precision, recall, \_ = precision\_recall\_curve(y\_true, y\_prob)  
 ax[2].plot(recall, precision, color='blue', lw=1.5)  
 ax[2].set\_xlabel('Recall', fontsize=8)  
 ax[2].set\_ylabel('Precision', fontsize=8)  
 ax[2].set\_title(f'Precision-Recall Curve', fontsize=10)  
 ax[2].tick\_params(labelsize=8)  
   
 plt.suptitle(f'{model\_name} Evaluation', fontsize=12)  
 plt.tight\_layout()  
 plt.subplots\_adjust(top=0.85)  
 plt.show()  
   
 return metrics  
  
# Dictionary to store metrics for comparison  
all\_metrics = {}  
  
# Evaluate all models  
rf\_metrics = evaluate\_model(y\_test, rf\_preds, rf\_probs, "Random Forest", all\_metrics)  
xgb\_metrics = evaluate\_model(y\_test, xgb\_preds, xgb\_probs, "XGBoost", all\_metrics)  
stacking\_metrics = evaluate\_model(y\_test, stacking\_preds, stacking\_probs, "Stacking Ensemble", all\_metrics)  
  
# Create comparison DataFrame  
comparison = pd.DataFrame(all\_metrics).T  
  
# Plot comparison as a single compact chart  
plt.figure(figsize=(8, 4))  
comparison.plot(kind='bar', figsize=(8, 4))  
plt.title('Model Performance Comparison')  
plt.ylabel('Score')  
plt.xticks(rotation=30)  
plt.legend(title='Metric', loc='upper center', bbox\_to\_anchor=(0.5, -0.15), ncol=4)  
plt.tight\_layout()  
plt.show()

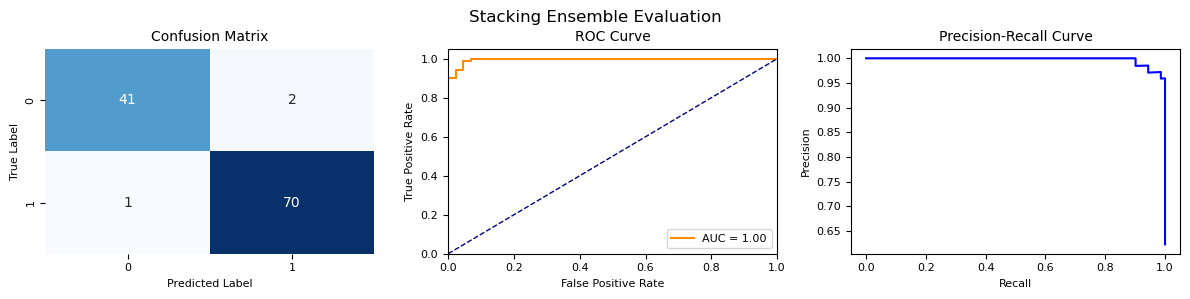
Random Forest: Acc=0.9561, Prec=0.9583, Rec=0.9718, F1=0.9650



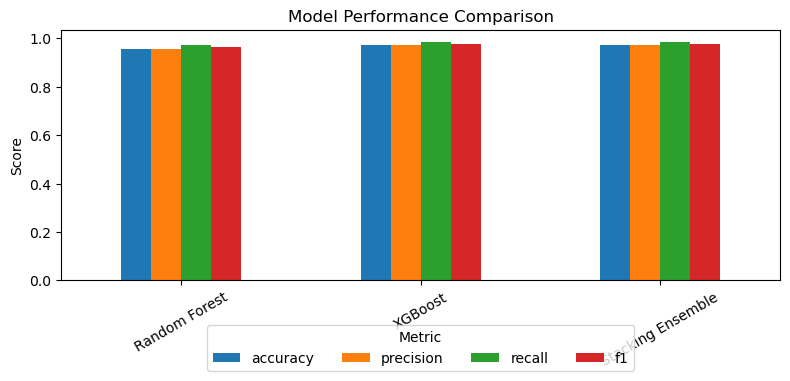
XGBoost: Acc=0.9737, Prec=0.9722, Rec=0.9859, F1=0.9790



Stacking Ensemble: Acc=0.9737, Prec=0.9722, Rec=0.9859, F1=0.9790



<Figure size 800x400 with 0 Axes>

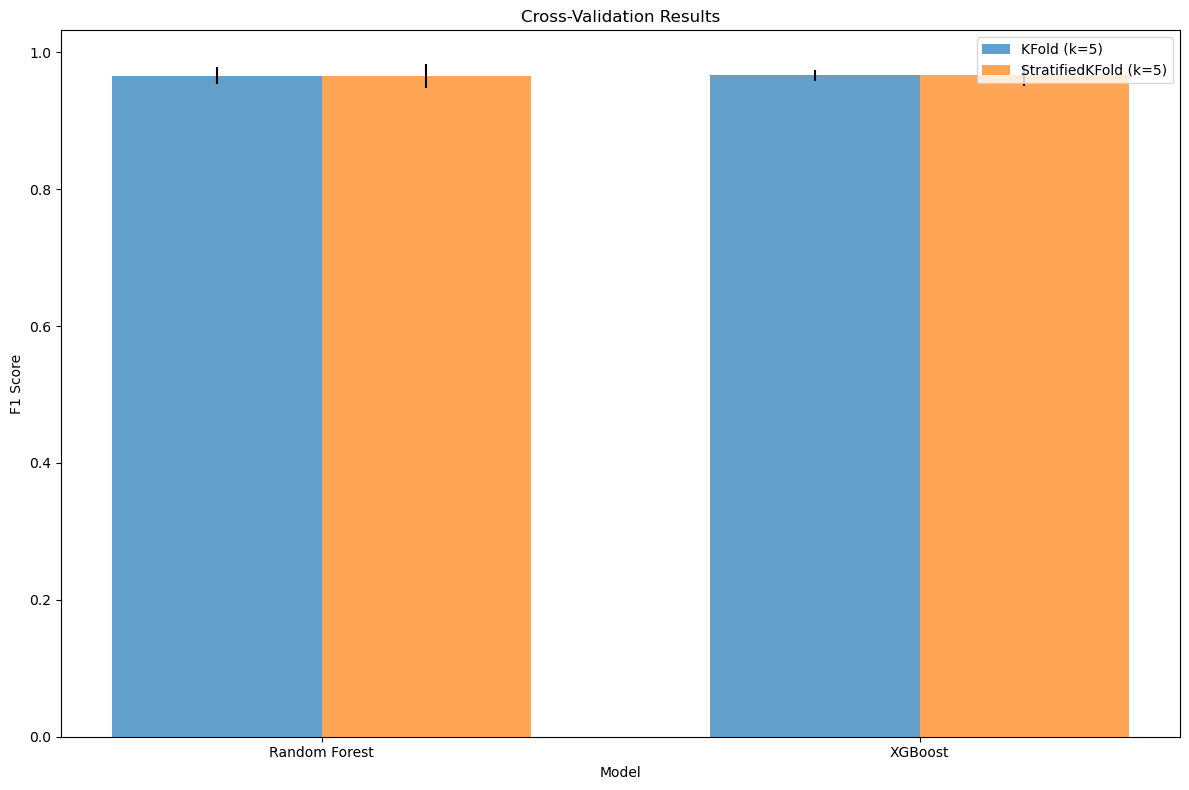


## 5. Cross-Validation Techniques

Let's implement different cross-validation strategies to evaluate our models more thoroughly:

# Define cross-validation strategies  
cv\_strategies = {  
 'KFold (k=5)': KFold(n\_splits=5, shuffle=True, random\_state=42),  
 'StratifiedKFold (k=5)': StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)  
}  
  
# Define models to evaluate  
models\_for\_cv = {  
 'Random Forest': RandomForestClassifier(n\_estimators=100, random\_state=42),  
 'XGBoost': xgb.XGBClassifier(  
 objective='binary:logistic',  
 max\_depth=6,  
 eta=0.3,  
 subsample=0.8,  
 colsample\_bytree=0.8,  
 random\_state=42  
 )  
}  
  
# Perform cross-validation  
cv\_results = {}  
for model\_name, model in models\_for\_cv.items():  
 cv\_results[model\_name] = {}  
 for cv\_name, cv in cv\_strategies.items():  
 scores = cross\_val\_score(model, X\_train\_scaled, y\_train, cv=cv, scoring='f1')  
 cv\_results[model\_name][cv\_name] = {  
 'mean': scores.mean(),  
 'std': scores.std(),  
 'scores': scores  
 }  
 print(f"{model\_name} with {cv\_name}: mean F1 = {scores.mean():.4f}, std = {scores.std():.4f}")  
  
# Visualize cross-validation results  
plt.figure(figsize=(12, 8))  
bar\_width = 0.35  
index = np.arange(len(models\_for\_cv))  
  
for i, cv\_name in enumerate(cv\_strategies.keys()):  
 means = [cv\_results[model\_name][cv\_name]['mean'] for model\_name in models\_for\_cv.keys()]  
 stds = [cv\_results[model\_name][cv\_name]['std'] for model\_name in models\_for\_cv.keys()]  
   
 plt.bar(index + i\*bar\_width, means, bar\_width, yerr=stds,   
 label=cv\_name, alpha=0.7)  
  
plt.xlabel('Model')  
plt.ylabel('F1 Score')  
plt.title('Cross-Validation Results')  
plt.xticks(index + bar\_width/2, models\_for\_cv.keys())  
plt.legend()  
plt.tight\_layout()  
plt.show()

Random Forest with KFold (k=5): mean F1 = 0.9660, std = 0.0131  
Random Forest with StratifiedKFold (k=5): mean F1 = 0.9652, std = 0.0178  
XGBoost with KFold (k=5): mean F1 = 0.9665, std = 0.0083  
XGBoost with StratifiedKFold (k=5): mean F1 = 0.9665, std = 0.0156

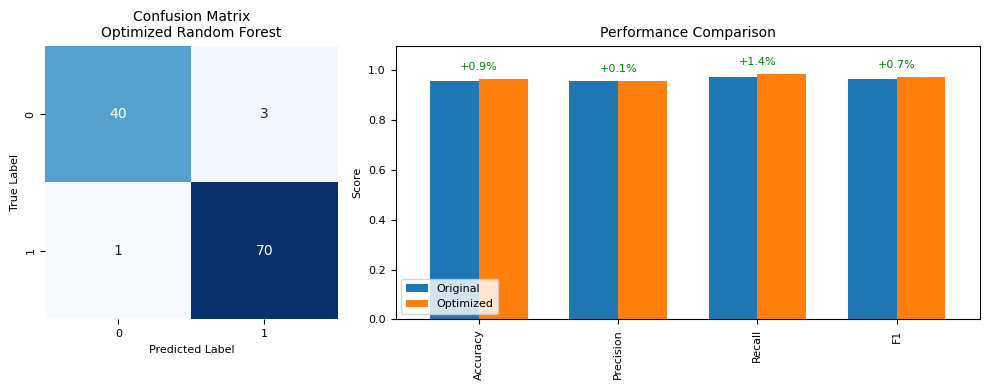


## 6. Advanced Optimization

Let's implement hyperparameter tuning to optimize our models:

from sklearn.model\_selection import GridSearchCV  
def tune\_and\_evaluate\_model(base\_model, param\_grid, X\_train, y\_train, X\_test, y\_test, model\_name, original\_metrics=None):  
 """  
 Compact function for model tuning and evaluation  
 """  
 # Create figure for results  
 fig, ax = plt.subplots(1, 2, figsize=(10, 4), gridspec\_kw={'width\_ratios': [1, 2]})  
   
 # Perform grid search  
 print(f"Tuning {model\_name}...")  
 grid\_search = GridSearchCV(  
 estimator=base\_model,  
 param\_grid=param\_grid,  
 cv=5,  
 scoring='f1',  
 n\_jobs=-1  
 )  
   
 grid\_search.fit(X\_train, y\_train)  
   
 # Get best model  
 best\_model = grid\_search.best\_estimator\_  
 print(f"Best parameters: {grid\_search.best\_params\_}")  
   
 # Evaluate optimized model  
 best\_preds = best\_model.predict(X\_test)  
 best\_probs = best\_model.predict\_proba(X\_test)[:, 1]  
   
 # Calculate metrics  
 best\_metrics = {  
 'accuracy': accuracy\_score(y\_test, best\_preds),  
 'precision': precision\_score(y\_test, best\_preds),  
 'recall': recall\_score(y\_test, best\_preds),  
 'f1': f1\_score(y\_test, best\_preds)  
 }  
   
 # Print metrics in compact format  
 print(f"Optimized {model\_name}: Acc={best\_metrics['accuracy']:.4f}, Prec={best\_metrics['precision']:.4f}, "  
 f"Rec={best\_metrics['recall']:.4f}, F1={best\_metrics['f1']:.4f}")  
   
 # Plot confusion matrix  
 cm = confusion\_matrix(y\_test, best\_preds)  
 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax[0], cbar=False)  
 ax[0].set\_title(f'Confusion Matrix\nOptimized {model\_name}', fontsize=10)  
 ax[0].set\_ylabel('True Label', fontsize=8)  
 ax[0].set\_xlabel('Predicted Label', fontsize=8)  
 ax[0].tick\_params(labelsize=8)  
   
 # Compare with original model if provided  
 if original\_metrics:  
 # Create comparison dataframe  
 metrics\_df = pd.DataFrame({  
 'Original': [original\_metrics['accuracy'], original\_metrics['precision'],   
 original\_metrics['recall'], original\_metrics['f1']],  
 'Optimized': [best\_metrics['accuracy'], best\_metrics['precision'],   
 best\_metrics['recall'], best\_metrics['f1']],  
 'Improvement': [best\_metrics['accuracy'] - original\_metrics['accuracy'],  
 best\_metrics['precision'] - original\_metrics['precision'],  
 best\_metrics['recall'] - original\_metrics['recall'],  
 best\_metrics['f1'] - original\_metrics['f1']]  
 }, index=['Accuracy', 'Precision', 'Recall', 'F1'])  
   
 # Print comparison  
 print("\nImprovement after optimization:")  
 for metric in ['accuracy', 'precision', 'recall', 'f1']:  
 improvement = best\_metrics[metric] - original\_metrics[metric]  
 print(f"{metric.capitalize()}: {improvement:.4f} ({improvement/original\_metrics[metric]\*100:.1f}%)")  
   
 # Plot comparison  
 metrics\_df[['Original', 'Optimized']].plot(kind='bar', ax=ax[1], width=0.7)  
   
 # Add text annotations for improvement percentages  
 for i, metric in enumerate(['accuracy', 'precision', 'recall', 'f1']):  
 improvement = best\_metrics[metric] - original\_metrics[metric]  
 percentage = improvement/original\_metrics[metric]\*100  
 ax[1].annotate(f"+{percentage:.1f}%",   
 xy=(i, best\_metrics[metric]),   
 xytext=(0, 5),  
 textcoords="offset points",   
 ha='center', va='bottom',  
 fontsize=8, color='green')  
   
 ax[1].set\_title(f'Performance Comparison', fontsize=10)  
 ax[1].set\_ylabel('Score', fontsize=8)  
 ax[1].set\_ylim(0, 1.1)  
 ax[1].legend(fontsize=8)  
 ax[1].tick\_params(labelsize=8)  
   
 plt.tight\_layout()  
 plt.show()  
   
 return best\_model, best\_metrics  
  
# Define parameter grid for Random Forest  
param\_grid = {  
 'n\_estimators': [3, 5, 10, 50, 100],  
 'max\_depth': [None, 2, 3, 5, 10, 20],  
 'min\_samples\_split': [2, 3, 5, 10]  
}  
  
# Tune and evaluate Random Forest  
best\_rf, best\_rf\_metrics = tune\_and\_evaluate\_model(  
 RandomForestClassifier(random\_state=42),  
 param\_grid,  
 X\_train\_scaled,   
 y\_train,  
 X\_test\_scaled,  
 y\_test,  
 "Random Forest",  
 rf\_metrics  
)

Tuning Random Forest...  
Best parameters: {'max\_depth': None, 'min\_samples\_split': 3, 'n\_estimators': 100}  
Optimized Random Forest: Acc=0.9649, Prec=0.9589, Rec=0.9859, F1=0.9722  
  
Improvement after optimization:  
Accuracy: 0.0088 (0.9%)  
Precision: 0.0006 (0.1%)  
Recall: 0.0141 (1.4%)  
F1: 0.0072 (0.7%)



## 7. Clean Up Resources

Always clean up resources to avoid unnecessary charges:

import boto3  
from botocore.exceptions import ClientError  
  
# Initialize the SageMaker client  
sagemaker\_client = boto3.client('sagemaker')  
  
# Function to delete all SageMaker models  
def delete\_all\_models():  
 try:  
 # List all models  
 models = sagemaker\_client.list\_models()  
 for model in models['Models']:  
 model\_name = model['ModelName']  
 print(f"Deleting model: {model\_name}")  
 sagemaker\_client.delete\_model(ModelName=model\_name)  
 print(f"Model {model\_name} deleted.")  
 except ClientError as e:  
 print(f"Error while deleting models: {e}")  
  
# Function to delete all SageMaker endpoints  
def delete\_all\_endpoints():  
 try:  
 # List all endpoints  
 endpoints = sagemaker\_client.list\_endpoints()  
 for endpoint in endpoints['Endpoints']:  
 endpoint\_name = endpoint['EndpointName']  
 print(f"Deleting endpoint: {endpoint\_name}")  
 sagemaker\_client.delete\_endpoint(EndpointName=endpoint\_name)  
 print(f"Endpoint {endpoint\_name} deleted.")  
 except ClientError as e:  
 print(f"Error while deleting endpoints: {e}")  
  
# Function to delete all SageMaker endpoint configurations  
def delete\_all\_endpoint\_configs():  
 try:  
 # List all endpoint configurations  
 endpoint\_configs = sagemaker\_client.list\_endpoint\_configs()  
 for config in endpoint\_configs['EndpointConfigs']:  
 config\_name = config['EndpointConfigName']  
 print(f"Deleting endpoint configuration: {config\_name}")  
 sagemaker\_client.delete\_endpoint\_config(EndpointConfigName=config\_name)  
 print(f"Endpoint configuration {config\_name} deleted.")  
 except ClientError as e:  
 print(f"Error while deleting endpoint configurations: {e}")  
  
# Clean up all SageMaker resources  
def clean\_up\_sagemaker\_resources():  
 delete\_all\_models()  
 delete\_all\_endpoints()  
 delete\_all\_endpoint\_configs()  
  
# Call the clean-up function  
clean\_up\_sagemaker\_resources()

## Common Mistakes and Best Practices

### Common Mistakes

* Not scaling features before training models that are sensitive to feature scales
* Using the test set for hyperparameter tuning, leading to data leakage
* Forgetting to clean up AWS resources after completing the lab
* Misinterpreting evaluation metrics (e.g., high accuracy with imbalanced data)
* Ignoring cross-validation in favor of a single train-test split

### Best Practices

* Always scale features when using distance-based algorithms or neural networks
* Use cross-validation for more reliable model evaluation
* Examine multiple evaluation metrics, not just accuracy
* Visualize model performance using confusion matrices, ROC curves, and PR curves
* Implement proper resource cleanup to avoid unnecessary AWS charges
* Start with simpler models before moving to more complex ones
* Understand the bias-variance tradeoff when optimizing models
* Document your modeling process and decisions

## Conclusion

In this lab, you've learned how to:

* Implement and compare different ensemble methods (bagging, boosting, and stacking)
* Create a comprehensive model evaluation framework
* Apply cross-validation techniques to validate model performance
* Optimize models through hyperparameter tuning
* Visualize and interpret model results