```
!pip install py7zr
```

```
Collecting py7zr
  Downloading py7zr-1.0.0-py3-none-any.whl.metadata (17 kB)
Collecting texttable (from py7zr)
  Downloading texttable-1.7.0-py2.py3-none-any.whl.metadata (9.8 kB)
Requirement already satisfied: pycryptodomex>=3.20.0 in /usr/local/lib/python3.12/dist-packages (from py7zr) (3.23.0)
Requirement already satisfied: brotli>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from py7zr) (1.1.0)
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (from py7zr) (5.9.5)
Collecting pyzstd>=0.16.1 (from py7zr)
  Downloading pyzstd-0.18.0-cp312-cp312-manylinux2014_x86_64.manylinux_2_17_x86_64.manylinux_2_28_x86_64.whl.metadata (2.6 kB)
Collecting pyppmd<1.3.0,>=1.1.0 (from py7zr)
  Downloading pyppmd-1.2.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (5.4 kB)
Collecting pybcj<1.1.0,>=1.0.0 (from py7zr)
  Downloading pybcj-1.0.6-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.7 kB)
Collecting multivolumefile>=0.2.3 (from py7zr)
  Downloading multivolumefile-0.2.3-py3-none-any.whl.metadata (6.3 kB)
Collecting inflate64<1.1.0,>=1.0.0 (from py7zr)
  Downloading inflate64-1.0.3-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.4 kB)
Requirement already satisfied: typing-extensions>=4.13.2 in /usr/local/lib/python3.12/dist-packages (from pyzstd>=0.16.1->py7zr)
Downloading py7zr-1.0.0-py3-none-any.whl (69 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 69.7/69.7 kB 6.4 MB/s eta 0:00:00
Downloading inflate64-1.0.3-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (97 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 97.0/97.0 kB 9.7 MB/s eta 0:00:00
Downloading multivolumefile-0.2.3-py3-none-any.whl (17 kB)
Downloading pybcj-1.0.6-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (51 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 51.7/51.7 kB 6.5 MB/s eta 0:00:00
Downloading pyppmd-1.2.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (142 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 142.7/142.7 kB 18.4 MB/s eta 0:00:00
Downloading pyzstd-0.18.0-cp312-cp312-manylinux2014_x86_64.manylinux_2_17_x86_64.manylinux_2_28_x86_64.whl (429 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 429.9/429.9 kB 42.1 MB/s eta 0:00:00
Downloading texttable-1.7.0-py2.py3-none-any.whl (10 kB)
Installing collected packages: texttable, pyzstd, pyppmd, pybcj, multivolumefile, inflate64, py7zr
Successfully installed inflate64-1.0.3 multivolumefile-0.2.3 py7zr-1.0.0 pybcj-1.0.6 pyppmd-1.2.0 pyzstd-0.18.0 texttable-1.7.0
```

```
import py7zr

archive_path = '/content/Churn-Prediction-Credit-Card-main.7z'
extract_path = '/content/'

with py7zr.SevenZipFile(archive_path, mode='r') as z:
    z.extractall(path=extract_path)
```

```
!pip install lightgbm xgboost catboost optuna scikit-learn pandas numpy matplotlib seaborn
```

```
Requirement already satisfied: lightgbm in /usr/local/lib/python3.12/dist-packages (4.6.0)
Requirement already satisfied: xgboost in /usr/local/lib/python3.12/dist-packages (3.0.5)
Collecting catboost
  Downloading catboost-1.2.8-cp312-cp312-manylinux2014_x86_64.whl.metadata (1.2 kB)
Collecting optuna
  Downloading optuna-4.5.0-py3-none-any.whl.metadata (17 kB)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.12/dist-packages (1.6.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages (2.2.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (2.0.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.12/dist-packages (0.13.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packages (from lightgbm) (1.16.2)
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.12/dist-packages (from xgboost) (2.27.3)
Requirement already satisfied: graphviz in /usr/local/lib/python3.12/dist-packages (from catboost) (0.21)
Requirement already satisfied: plotly in /usr/local/lib/python3.12/dist-packages (from catboost) (5.24.1)
Requirement already satisfied: six in /usr/local/lib/python3.12/dist-packages (from catboost) (1.17.0)
Requirement already satisfied: alembic>=1.5.0 in /usr/local/lib/python3.12/dist-packages (from optuna) (1.17.0)
Collecting colorlog (from optuna)
  Downloading colorlog-6.10.1-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from optuna) (25.0)
Requirement already satisfied: sqlalchemy>=1.4.2 in /usr/local/lib/python3.12/dist-packages (from optuna) (2.0.44)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from optuna) (4.67.1)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.12/dist-packages (from optuna) (6.0.3)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (3.6.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.12/dist-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (4.60.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.4.9)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (3.2.5)
Requirement already satisfied: Mako in /usr/local/lib/python3.12/dist-packages (from alembic>=1.5.0->optuna) (1.3.10)
Requirement already satisfied: typing-extensions>=4.12 in /usr/local/lib/python3.12/dist-packages (from alembic>=1.5.0->optuna)
Requirement already satisfied: greenlet>=1 in /usr/local/lib/python3.12/dist-packages (from sqlalchemy>=1.4.2->optuna) (3.2.4)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.12/dist-packages (from plotly->catboost) (8.5.0)
Requirement already satisfied: MarkupSafe>=0.9.2 in /usr/local/lib/python3.12/dist-packages (from Mako->alembic>=1.5.0->optuna)
```

```
Downloading catboost-1.2.8-cp312-cp312-manylinux2014_x86_64.whl (99.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 99.2/99.2 MB 10.2 MB/s eta 0:00:00
Downloading optuna-4.5.0-py3-none-any.whl (400 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 400.9/400.9 kB 28.3 MB/s eta 0:00:00
Downloading colorlog-6.10.1-py3-none-any.whl (11 kB)
Installing collected packages: colorlog, optuna, catboost
Successfully installed catboost-1.2.8 colorlog-6.10.1 optuna-4.5.0
```

```
print("🔧 FIXED Model Training - No Target Leakage")
print("🎯 Implementing proper ML validation practices")
print("=" * 60)

import pandas as pd
import numpy as np
import pickle
import json
import os
import sys
from datetime import datetime
import time
import warnings
warnings.filterwarnings('ignore')

# Set working directory for Colab (uncomment for Colab)
os.chdir('/content/Churn-Prediction-Credit-Card-main/notebooks')
print(f"📁 Working directory: {os.getcwd()}")

# Add project root to path
sys.path.append('../')

# Core ML libraries
from sklearn.model_selection import (
    train_test_split, StratifiedKFold, cross_val_score,
    validation_curve, GridSearchCV, cross_validate
)
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, classification_report, confusion_matrix,
    roc_curve, precision_recall_curve
)

# Advanced ML libraries
try:
    import lightgbm as lgb
    print("✅ LightGBM available")
except ImportError:
    print("⚠️ LightGBM not available")
    lgb = None

try:
    import xgboost as xgb
    print("✅ XGBoost available")
except ImportError:
    print("⚠️ XGBoost not available")
    xgb = None

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Custom components
from src.exception import CustomException
from src.logger import logging

print(f"⏰ Environment loaded at: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print("✅ Ready to load CLEAN data without target leakage!\n")
```

```
🔧 FIXED Model Training - No Target Leakage
🎯 Implementing proper ML validation practices
============================================================
📁 Working directory: /content/Churn-Prediction-Credit-Card-main/notebooks
✅ LightGBM available
✅ XGBoost available
⏰ Environment loaded at: 2025-10-20 12:47:52
✅ Ready to load CLEAN data without target leakage!
```

```
print("🧹 Loading and Cleaning Data (Removing Target Leakage)...")
print("=" * 55)
```

```
def load_clean_data():
    """Load original data and remove target leakage columns."""
    try:
        # Load original raw data
        print("📂 Loading original dataset...")
        data_path = '../input/BankChurners.csv'
        if not os.path.exists(data_path):
            data_path = 'input/BankChurners.csv'

        df = pd.read_csv(data_path)
        print(f"   Original data shape: {df.shape}")
        print(f"   Columns: {list(df.columns)}")

        # CRITICAL: Identify and remove target leakage columns
        print("\n🚨 Identifying target leakage columns...")
        leakage_patterns = ['naive_bayes', 'classifier', 'prediction', 'prob']
        leakage_columns = []

        for col in df.columns:
            col_lower = col.lower()
            if any(pattern in col_lower for pattern in leakage_patterns):
                leakage_columns.append(col)

        if leakage_columns:
            print(f"🔥 FOUND TARGET LEAKAGE COLUMNS:")
            for col in leakage_columns:
                print(f"   ❌ {col}")

            print(f"\n💥 REMOVING {len(leakage_columns)} leakage columns...")
            df_clean = df.drop(columns=leakage_columns)
            print(f"   Clean data shape: {df_clean.shape}")
        else:
            print("✅ No obvious target leakage columns found")
            df_clean = df.copy()

        # Prepare target variable
        print(f"\n🎯 Preparing target variable...")
        df_clean.columns = [x.lower().replace(' ', '_') for x in df_clean.columns]

        # Handle target column
        target_col = 'attrition_flag'
        if target_col in df_clean.columns:
            df_clean['churn_flag'] = df_clean[target_col].map({
                'Attrited Customer': 1,
                'Existing Customer': 0
            })
            df_clean = df_clean.drop(columns=[target_col])

        # Remove identifier columns
        id_columns = ['clientnum']
        existing_id_cols = [col for col in id_columns if col in df_clean.columns]
        if existing_id_cols:
            print(f"🗑️ Removing ID columns: {existing_id_cols}")
            df_clean = df_clean.drop(columns=existing_id_cols)

        # Final validation
        print(f"\n✅ CLEAN DATASET READY:")
        print(f"   Shape: {df_clean.shape}")
        print(f"   Features: {df_clean.shape[1] - 1}")  # -1 for target
        print(f"   Target distribution:")
        print(f"     No Churn (0): {(df_clean['churn_flag'] == 0).sum()}")
        print(f"     Churn (1): {(df_clean['churn_flag'] == 1).sum()}")

        return df_clean

    except Exception as e:
        raise CustomException(e, sys)

# Load clean data
df_clean = load_clean_data()
```

```
🧹 Loading and Cleaning Data (Removing Target Leakage)...
======================================================
📂 Loading original dataset...
   Original data shape: (10127, 23)
   Columns: ['CLIENTNUM', 'Attrition_Flag', 'Customer_Age', 'Gender', 'Dependent_count', 'Education_Level', 'Marital_Status', 'I

🚨 Identifying target leakage columns...
🔥 FOUND TARGET LEAKAGE COLUMNS:
   ❌ Naive_Bayes_Classifier_Attrition_Flag_Card_Category_Contacts_Count_12_mon_Dependent_count_Education_Level_Months_Inactive_
   ❌ Naive_Bayes_Classifier_Attrition_Flag_Card_Category_Contacts_Count_12_mon_Dependent_count_Education_Level_Months_Inactive_
```

```
💥 REMOVING 2 leakage columns...
   Clean data shape: (10127, 21)

🎯 Preparing target variable...
🗑️ Removing ID columns: ['clientnum']

✅ CLEAN DATASET READY:
   Shape: (10127, 20)
   Features: 19
   Target distribution:
     No Churn (0): 8500
     Churn (1): 1627
```

```
print("\n🔀 Creating Proper Train-Validation-Test Split...")
print("=" * 50)

def create_proper_splits(df, test_size=0.2, val_size=0.2, random_state=42):
    """Create proper train/validation/test splits to prevent data leakage."""

    try:
        # Separate features and target
        X = df.drop('churn_flag', axis=1)
        y = df['churn_flag']

        print(f"📊 Original data: {X.shape[0]} samples, {X.shape[1]} features")

        # First split: separate test set (20%)
        X_temp, X_test, y_temp, y_test = train_test_split(
            X, y,
            test_size=test_size,
            random_state=random_state,
            stratify=y
        )

        # Second split: separate train and validation from remaining 80%
        val_size_adjusted = val_size / (1 - test_size)  # Adjust for remaining data
        X_train, X_val, y_train, y_val = train_test_split(
            X_temp, y_temp,
            test_size=val_size_adjusted,
            random_state=random_state,
            stratify=y_temp
        )

        print(f"🎯 Data splits created:")
        print(f"   Training:   {X_train.shape[0]} samples ({len(X_train)/len(X)*100:.1f}%)")
        print(f"   Validation: {X_val.shape[0]} samples ({len(X_val)/len(X)*100:.1f}%)")
        print(f"   Test:       {X_test.shape[0]} samples ({len(X_test)/len(X)*100:.1f}%)")

        # Verify target distribution
        print(f"\n📈 Target distribution:")
        for split_name, y_split in [('Train', y_train), ('Val', y_val), ('Test', y_test)]:
            churn_rate = (y_split == 1).mean()
            print(f"   {split_name}: {churn_rate:.3f} churn rate")

        # Check for data leakage (should be no overlap)
        train_index = set(X_train.index)
        val_index = set(X_val.index)
        test_index = set(X_test.index)

        assert len(train_index & val_index) == 0, "LEAKAGE: Train-Val overlap"
        assert len(train_index & test_index) == 0, "LEAKAGE: Train-Test overlap"
        assert len(val_index & test_index) == 0, "LEAKAGE: Val-Test overlap"

        print("✅ No data leakage between splits confirmed")

        return X_train, X_val, X_test, y_train, y_val, y_test

    except Exception as e:
        raise CustomException(e, sys)

# Create splits
X_train, X_val, X_test, y_train, y_val, y_test = create_proper_splits(df_clean)
```

```
🔀 Creating Proper Train-Validation-Test Split...
==================================================
📊 Original data: 10127 samples, 19 features
🎯 Data splits created:
   Training:   6075 samples (60.0%)
   Validation: 2026 samples (20.0%)
   Test:       2026 samples (20.0%)
```

```
📈 Target distribution:
   Train: 0.161 churn rate
   Val: 0.161 churn rate
   Test: 0.160 churn rate
✅ No data leakage between splits confirmed
```

```
print("\n🔧 Feature Engineering (Training Data Only)...")
print("=" * 45)

def safe_feature_engineering(X_train, X_val, X_test):
    """Apply feature engineering without target leakage."""

    try:
        from sklearn.preprocessing import StandardScaler, LabelEncoder

        # Identify categorical and numerical columns
        categorical_cols = X_train.select_dtypes(include=['object']).columns.tolist()
        numerical_cols = X_train.select_dtypes(include=[np.number]).columns.tolist()

        print(f"📊 Feature types identified:")
        print(f"   Categorical: {len(categorical_cols)} columns")
        print(f"   Numerical: {len(numerical_cols)} columns")

        # STEP 1: Handle categorical variables (fit only on training data)
        print(f"\n🔤 Encoding categorical variables...")

        # Create copies to avoid modifying originals
        X_train_processed = X_train.copy()
        X_val_processed = X_val.copy()
        X_test_processed = X_test.copy()

        label_encoders = {}

        for col in categorical_cols:
            print(f"   Encoding: {col}")
            le = LabelEncoder()

            # Fit only on training data
            X_train_processed[col] = le.fit_transform(X_train_processed[col].astype(str))

            # Transform validation and test (may have unseen categories)
            try:
                X_val_processed[col] = le.transform(X_val_processed[col].astype(str))
                X_test_processed[col] = le.transform(X_test_processed[col].astype(str))
            except ValueError as e:
                # Handle unseen categories by assigning them to most frequent class
                print(f"     Warning: Unseen categories in {col}, using most frequent class")
                most_frequent = X_train[col].mode()[0] if len(X_train[col].mode()) > 0 else 'Unknown'
                most_frequent_encoded = le.transform([most_frequent])[0]

                # Replace unseen categories
                val_mask = ~X_val_processed[col].astype(str).isin(le.classes_)
                test_mask = ~X_test_processed[col].astype(str).isin(le.classes_)

                X_val_processed.loc[val_mask, col] = most_frequent_encoded
                X_test_processed.loc[test_mask, col] = most_frequent_encoded

                # Now apply transform
                X_val_processed[col] = le.transform(X_val_processed[col].astype(str))
                X_test_processed[col] = le.transform(X_test_processed[col].astype(str))

            label_encoders[col] = le

        # STEP 2: Scale numerical features (fit only on training data)
        print(f"\n📏 Scaling numerical features...")

        if numerical_cols:
            scaler = StandardScaler()

            # Fit only on training data
            X_train_processed[numerical_cols] = scaler.fit_transform(X_train_processed[numerical_cols])

            # Transform validation and test
            X_val_processed[numerical_cols] = scaler.transform(X_val_processed[numerical_cols])
            X_test_processed[numerical_cols] = scaler.transform(X_test_processed[numerical_cols])

            print(f"   Scaled {len(numerical_cols)} numerical features")

        print(f"\n✅ Feature engineering complete:")
        print(f"   Train shape: {X_train_processed.shape}")
        print(f"   Val shape: {X_val_processed.shape}")
```

```
            print(f"   Test shape: {X_test_processed.shape}")

            return X_train_processed, X_val_processed, X_test_processed, label_encoders

        except Exception as e:
            raise CustomException(e, sys)

    # Apply feature engineering
    X_train_processed, X_val_processed, X_test_processed, label_encoders = safe_feature_engineering(
        X_train, X_val, X_test
    )
```

```
🔧 Feature Engineering (Training Data Only)...
==================================================
📊 Feature types identified:
   Categorical: 5 columns
   Numerical: 14 columns

🔤 Encoding categorical variables...
   Encoding: gender
   Encoding: education_level
   Encoding: marital_status
   Encoding: income_category
   Encoding: card_category

📏 Scaling numerical features...
   Scaled 14 numerical features

✅ Feature engineering complete:
   Train shape: (6075, 19)
   Val shape: (2026, 19)
   Test shape: (2026, 19)
```

```
print("\n🤖 Model Training with Cross-Validation...")
print("=" * 45)

def train_models_with_validation(X_train, y_train, X_val, y_val):
    """Train models with proper validation to prevent overfitting."""

    models = {}
    results = {}
    cv_results = {}

    # Use StratifiedKFold for cross-validation
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

    print("🔥 Training models with 5-fold cross-validation...")

    # 1. Logistic Regression with regularization
    print("\n📊 1. Training Logistic Regression...")
    lr = LogisticRegression(
        random_state=42,
        max_iter=1000,
        C=1.0,  # L2 regularization
        class_weight='balanced'  # Handle class imbalance
    )

    # Cross-validation on training set
    cv_scores = cross_val_score(lr, X_train, y_train, cv=cv, scoring='roc_auc')
    cv_results['Logistic Regression'] = {
        'cv_mean': cv_scores.mean(),
        'cv_std': cv_scores.std(),
        'cv_scores': cv_scores.tolist()
    }

    # Fit on training and evaluate on validation
    lr.fit(X_train, y_train)
    lr_val_pred = lr.predict(X_val)
    lr_val_proba = lr.predict_proba(X_val)[:, 1]

    models['Logistic Regression'] = lr
    results['Logistic Regression'] = {
        'val_accuracy': accuracy_score(y_val, lr_val_pred),
        'val_precision': precision_score(y_val, lr_val_pred),
        'val_recall': recall_score(y_val, lr_val_pred),
        'val_f1': f1_score(y_val, lr_val_pred),
        'val_auc': roc_auc_score(y_val, lr_val_proba)
    }

    # 2. Random Forest with regularization
    print("🌳 2. Training Random Forest...")
    rf = RandomForestClassifier(
```

```
    n_estimators=100,  # Reduced to prevent overfitting
    max_depth=10,      # Limit depth
    min_samples_split=20,  # Require more samples to split
    min_samples_leaf=10,   # Require more samples in leaf
    random_state=42,
    class_weight='balanced',
    n_jobs=-1
)

# Cross-validation
cv_scores = cross_val_score(rf, X_train, y_train, cv=cv, scoring='roc_auc')
cv_results['Random Forest'] = {
    'cv_mean': cv_scores.mean(),
    'cv_std': cv_scores.std(),
    'cv_scores': cv_scores.tolist()
}

# Validation
rf.fit(X_train, y_train)
rf_val_pred = rf.predict(X_val)
rf_val_proba = rf.predict_proba(X_val)[:, 1]

models['Random Forest'] = rf
results['Random Forest'] = {
    'val_accuracy': accuracy_score(y_val, rf_val_pred),
    'val_precision': precision_score(y_val, rf_val_pred),
    'val_recall': recall_score(y_val, rf_val_pred),
    'val_f1': f1_score(y_val, rf_val_pred),
    'val_auc': roc_auc_score(y_val, rf_val_proba)
}

# 3. Gradient Boosting with regularization
print("⚡ 3. Training Gradient Boosting...")
gb = GradientBoostingClassifier(
    n_estimators=100,
    learning_rate=0.1,   # Moderate learning rate
    max_depth=5,         # Shallow trees
    min_samples_split=20,
    min_samples_leaf=10,
    subsample=0.8,       # Stochastic gradient boosting
    random_state=42
)

# Cross-validation
cv_scores = cross_val_score(gb, X_train, y_train, cv=cv, scoring='roc_auc')
cv_results['Gradient Boosting'] = {
    'cv_mean': cv_scores.mean(),
    'cv_std': cv_scores.std(),
    'cv_scores': cv_scores.tolist()
}

# Validation
gb.fit(X_train, y_train)
gb_val_pred = gb.predict(X_val)
gb_val_proba = gb.predict_proba(X_val)[:, 1]

models['Gradient Boosting'] = gb
results['Gradient Boosting'] = {
    'val_accuracy': accuracy_score(y_val, gb_val_pred),
    'val_precision': precision_score(y_val, gb_val_pred),
    'val_recall': recall_score(y_val, gb_val_pred),
    'val_f1': f1_score(y_val, gb_val_pred),
    'val_auc': roc_auc_score(y_val, gb_val_proba)
}

# 4. LightGBM (if available)
if lgb is not None:
    print("💡 4. Training LightGBM...")
    lgbm = lgb.LGBMClassifier(
        n_estimators=100,
        learning_rate=0.1,
        max_depth=5,
        min_child_samples=20,
        subsample=0.8,
        colsample_bytree=0.8,
        reg_alpha=0.1,      # L1 regularization
        reg_lambda=0.1,     # L2 regularization
        class_weight='balanced',
        random_state=42,
        verbose=-1
    )
```

```
        # Cross-validation
        cv_scores = cross_val_score(lgbm, X_train, y_train, cv=cv, scoring='roc_auc')
        cv_results['LightGBM'] = {
            'cv_mean': cv_scores.mean(),
            'cv_std': cv_scores.std(),
            'cv_scores': cv_scores.tolist()
        }

        # Validation
        lgbm.fit(X_train, y_train)
        lgbm_val_pred = lgbm.predict(X_val)
        lgbm_val_proba = lgbm.predict_proba(X_val)[:, 1]

        models['LightGBM'] = lgbm
        results['LightGBM'] = {
            'val_accuracy': accuracy_score(y_val, lgbm_val_pred),
            'val_precision': precision_score(y_val, lgbm_val_pred),
            'val_recall': recall_score(y_val, lgbm_val_pred),
            'val_f1': f1_score(y_val, lgbm_val_pred),
            'val_auc': roc_auc_score(y_val, lgbm_val_proba)
        }

    return models, results, cv_results

# Train models
start_time = time.time()
models, val_results, cv_results = train_models_with_validation(
    X_train_processed, y_train, X_val_processed, y_val
)
training_time = time.time() - start_time

print(f"\n⏱️ Training completed in {training_time:.2f} seconds")
```

```
🚂 Model Training with Cross-Validation...
==================================================
🔥 Training models with 5-fold cross-validation...

📊 1. Training Logistic Regression...
🌳 2. Training Random Forest...
⚡ 3. Training Gradient Boosting...
💡 4. Training LightGBM...

⏱️ Training completed in 18.91 seconds
```

```
print("\n📊 Model Performance Analysis (Cross-Validation + Validation)...")
print("=" * 65)

def analyze_realistic_performance(val_results, cv_results):
    """Analyze model performance showing realistic results."""

    print("🎯 CROSS-VALIDATION RESULTS (5-Fold):")
    print("=" * 40)

    for model_name, cv_res in cv_results.items():
        print(f"\n{model_name}:")
        print(f"   CV AUC: {cv_res['cv_mean']:.4f} ± {cv_res['cv_std']:.4f}")
        print(f"   CV Range: [{cv_res['cv_mean'] - cv_res['cv_std']:.4f}, {cv_res['cv_mean'] + cv_res['cv_std']:.4f}]")

    print(f"\n\n🎯 VALIDATION SET RESULTS:")
    print("=" * 30)

    # Create performance DataFrame
    perf_df = pd.DataFrame(val_results).T
    perf_df = perf_df.round(4)

    # Sort by validation AUC
    perf_df_sorted = perf_df.sort_values('val_auc', ascending=False)

    for i, (model, metrics) in enumerate(perf_df_sorted.iterrows(), 1):
        print(f"\n{i}. {model}")
        print(f"   Val AUC: {metrics['val_auc']:.4f}")
        print(f"   Val Accuracy: {metrics['val_accuracy']:.4f}")
        print(f"   Val Precision: {metrics['val_precision']:.4f}")
        print(f"   Val Recall: {metrics['val_recall']:.4f}")
        print(f"   Val F1: {metrics['val_f1']:.4f}")

    # Check for overfitting
    print(f"\n🔍 OVERFITTING ANALYSIS:")
    print("=" * 25)

    for model_name in cv_results.keys():
```

```
        if model_name in val_results:
            cv_auc = cv_results[model_name]['cv_mean']
            val_auc = val_results[model_name]['val_auc']
            difference = cv_auc - val_auc

            print(f"\n{model_name}:")
            print(f"   CV AUC: {cv_auc:.4f}")
            print(f"   Val AUC: {val_auc:.4f}")
            print(f"   Difference: {difference:.4f}", end="")

            if abs(difference) < 0.02:
                print(" ✅ Good generalization")
            elif difference > 0.05:
                print(" ⚠️ Possible overfitting")
            else:
                print(" 🔄 Acceptable")

    # Find best model
    best_model_name = perf_df_sorted.index[0]
    best_metrics = perf_df_sorted.iloc[0]

    return perf_df_sorted, best_model_name, best_metrics

# Analyze performance
perf_df, best_model_name, best_metrics = analyze_realistic_performance(val_results, cv_results)
```

```
LightGBM:
   CV AUC: 0.9923 ± 0.0016
   CV Range: [0.9908, 0.9939]


🎯 VALIDATION SET RESULTS:
==============================

1. LightGBM
   Val AUC: 0.9910
   Val Accuracy: 0.9635
   Val Precision: 0.8539
   Val Recall: 0.9325
   Val F1: 0.8915

2. Gradient Boosting
   Val AUC: 0.9905
   Val Accuracy: 0.9684
   Val Precision: 0.9367
   Val Recall: 0.8620
   Val F1: 0.8978

3. Random Forest
   Val AUC: 0.9813
   Val Accuracy: 0.9432
   Val Precision: 0.7784
   Val Recall: 0.9049
   Val F1: 0.8369

4. Logistic Regression
   Val AUC: 0.9196
   Val Accuracy: 0.8342
   Val Precision: 0.4908
   Val Recall: 0.8190
   Val F1: 0.6138

🔍 OVERFITTING ANALYSIS:
========================

Logistic Regression:
   CV AUC: 0.9255
   Val AUC: 0.9196
   Difference: 0.0060 ✅ Good generalization

Random Forest:
   CV AUC: 0.9827
   Val AUC: 0.9813
   Difference: 0.0014 ✅ Good generalization

Gradient Boosting:
   CV AUC: 0.9919
   Val AUC: 0.9905
   Difference: 0.0014 ✅ Good generalization

LightGBM:
   CV AUC: 0.9923
   Val AUC: 0.9910
   Difference: 0.0014 ✅ Good generalization
```

```
print(f"\n🏆 Final Test Set Evaluation (Best Model: {best_model_name})...")
print("=" * 60)

def final_test_evaluation(best_model, X_test, y_test, model_name):
    """Final unbiased evaluation on test set."""

    try:
        print(f"🎯 Evaluating {model_name} on unseen test data...")

        # Get predictions
        test_pred = best_model.predict(X_test)
        test_proba = best_model.predict_proba(X_test)[:, 1]

        # Calculate metrics
        test_results = {
            'accuracy': accuracy_score(y_test, test_pred),
            'precision': precision_score(y_test, test_pred),
            'recall': recall_score(y_test, test_pred),
            'f1': f1_score(y_test, test_pred),
            'auc': roc_auc_score(y_test, test_proba)
        }

        print(f"\n🎉 FINAL TEST RESULTS:")
        print(f"=" * 25)
        print(f"Model: {model_name}")
        print(f"Test AUC: {test_results['auc']:.4f}")
        print(f"Test Accuracy: {test_results['accuracy']:.4f}")
        print(f"Test Precision: {test_results['precision']:.4f}")
        print(f"Test Recall: {test_results['recall']:.4f}")
        print(f"Test F1-Score: {test_results['f1']:.4f}")

        # Confusion matrix
        cm = confusion_matrix(y_test, test_pred)
        print(f"\n📊 Confusion Matrix:")
        print(f"True Negatives: {cm[0,0]}")
        print(f"False Positives: {cm[0,1]}")
        print(f"False Negatives: {cm[1,0]}")
        print(f"True Positives: {cm[1,1]}")

        return test_results

    except Exception as e:
        raise CustomException(e, sys)

# Final evaluation
best_model = models[best_model_name]
test_results = final_test_evaluation(
    best_model, X_test_processed, y_test, best_model_name
)
```

```
🏆 Final Test Set Evaluation (Best Model: LightGBM)...
============================================================
🎯 Evaluating LightGBM on unseen test data...

🎉 FINAL TEST RESULTS:
========================
Model: LightGBM
Test AUC: 0.9903
Test Accuracy: 0.9605
Test Precision: 0.8530
Test Recall: 0.9108
Test F1-Score: 0.8810

📊 Confusion Matrix:
True Negatives: 1650
False Positives: 51
False Negatives: 29
True Positives: 296
```

```
print(f"\n💾 Saving Clean Results...")
print("=" * 30)

def save_clean_results():
    """Save all results without target leakage."""

    try:
        # Set the base results directory
        results_dir = '/content/results'

        # Ensure the results directory exists
        os.makedirs(results_dir, exist_ok=True)
```

```
        print(f"✅ Ensured results directory exists: {results_dir}")

        # Save best model
        model_path = os.path.join(results_dir, 'clean_best_model.pkl')
        with open(model_path, 'wb') as f:
            pickle.dump(best_model, f)
        print(f"✅ Saved: {model_path}")

        # Save comprehensive results
        clean_results = {
            'experiment_info': {
                'timestamp': datetime.now().isoformat(),
                'target_leakage_removed': True,
                'proper_validation': True,
                'best_model': best_model_name
            },
            'data_info': {
                'original_features': len(df_clean.columns) - 1,
                'samples': {
                    'train': len(X_train),
                    'validation': len(X_val),
                    'test': len(X_test)
                },
                'target_distribution': {
                    'no_churn': int((y_train == 0).sum()),
                    'churn': int((y_train == 1).sum())
                }
            },
            'cross_validation_results': cv_results,
            'validation_results': {k: {k2: float(v2) for k2, v2 in v.items()} for k, v in val_results.items()},
            'final_test_results': {k: float(v) for k, v in test_results.items()},
            'model_comparison': {
                'best_model': best_model_name,
                'performance_ranking': perf_df.to_dict('index')
            }
        }

        results_path = os.path.join(results_dir, 'clean_training_results.json')
        with open(results_path, 'w') as f:
            json.dump(clean_results, f, indent=2)
        print(f"✅ Saved: {results_path}")

        return clean_results

    except Exception as e:
        raise CustomException(e, sys)

# Save results
clean_results = save_clean_results()

# Final Summary
print(f"\n🎉 CLEAN MODEL TRAINING COMPLETED!")
print("=" * 40)
print(f"✅ Target leakage eliminated")
print(f"✅ Proper train/val/test splits")
print(f"✅ Cross-validation implemented")
print(f"✅ Regularization applied")
print(f"✅ Realistic performance metrics")
print(f"\n🏆 Best Model: {best_model_name}")
print(f"🎯 Test AUC: {test_results['auc']:.4f} (Realistic!)")
print(f"📊 Test Accuracy: {test_results['accuracy']:.4f}")
print(f"\n📁 Clean artifacts saved:")
print(f"   ✅ clean_best_model.pkl")
print(f"   ✅ clean_training_results.json")
print(f"\n🚀 Ready for production deployment!")
```

```
💾 Saving Clean Results...
==============================
✅ Ensured results directory exists: /content/results
✅ Saved: /content/results/clean_best_model.pkl
✅ Saved: /content/results/clean_training_results.json

🎉 CLEAN MODEL TRAINING COMPLETED!
========================================
✅ Target leakage eliminated
✅ Proper train/val/test splits
✅ Cross-validation implemented
✅ Regularization applied
✅ Realistic performance metrics

🏆 Best Model: LightGBM
🎯 Test AUC: 0.9903 (Realistic!)
📊 Test Accuracy: 0.9605
```

```
📁 Clean artifacts saved:
   ✅ clean_best_model.pkl
   ✅ clean_training_results.json

🚀 Ready for production deployment!
```