```
# --- 1. Setup ---
import os
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, count, lit, rand, expr, min as pyspark_min, max as pyspark_max # Added min, max explicitly
from pyspark.sql.types import StringType, IntegerType, DoubleType, DateType, LongType, StructType, StructField # Ensure DateType is here
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer, IndexToString
from pyspark.ml.classification import LogisticRegression, RandomForestClassifier, GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# --- Mount Google Drive (if using Google Colab) ---
try:
    from google.colab import drive
    drive.mount('/content/drive')
    print("Google Drive mounted successfully.")
    google_drive_base_path = '/content/drive/MyDrive/'
except ImportError:
    print("Not running in Google Colab. Assuming local file system.")
    google_drive_base_path = ""

# Initialize SparkSession
spark = SparkSession.builder.appName("ReligareChurnModeling") \
    .config("spark.sql.legacy.timeParserPolicy", "LEGACY") \
    .config("spark.sql.shuffle.partitions", "100") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()

# Define ABT path
abt_output_dir = os.path.join(google_drive_base_path, 'Tables/output_abt_final_pred/')
abt_filename = "predictive_abt_religare_churn_2021_2023.parquet"
abt_path = os.path.join(abt_output_dir, abt_filename)

print("Setup Complete.")
print(f"ABT Path: {abt_path}")
```

```
Mounted at /content/drive
Google Drive mounted successfully.
Setup Complete.
ABT Path: /content/drive/MyDrive/Tables/output_abt_final_pred/predictive_abt_religare_churn_2021_2023.parquet
```

```
# --- 2. Load ABT and Initial Exploration ---
print(f"\nLoading ABT from: {abt_path}")
try:
    abt_df = spark.read.parquet(abt_path)
    # abt_df.persist() # <<--- DO NOT PERSIST THE FULL ABT HERE IF IT'S TOO LARGE

    print(f"ABT loaded. Verifying with a sample. First 10 columns, 3 rows:")
    if abt_df: # Check if DataFrame object exists
        abt_df.select(abt_df.columns[:10]).show(3, vertical=True, truncate=False)
        abt_df_column_count = len(abt_df.columns)
        print(f"ABT has {abt_df_column_count} columns.")
        # Getting full count here can trigger OOM if abt_df is huge and not persisted
        # print(f"Attempting to get row count (might be slow)...")
        # abt_df_row_count = abt_df.count()
        # print(f"ABT has {abt_df_row_count} rows and {abt_df_column_count} columns.")
    else:
        print("abt_df is None after read. Check path and file.")

    print("\nABT Schema:")
    if abt_df: abt_df.printSchema()

    print("\nDescriptive statistics for Tenure_Days and a few features (if ABT loaded):")
    if abt_df:
        desc_cols = ["Tenure_Days", "Trade_Days_Count_90D", "Login_Days_Count_90D", "Trade_Sum_90D"]
        existing_desc_cols = [c for c in desc_cols if c in abt_df.columns]
        if existing_desc_cols:
            abt_df.select(existing_desc_cols).describe().show()
        else:
            print("Selected descriptive columns not found for describe().")

except Exception as e:
    print(f"Error loading or exploring ABT: {e}")
```

```
Loading ABT from: /content/drive/MyDrive/Tables/output_abt_final_pred/predictive_abt_religare_churn_2021_2023.parquet
ABT loaded. Verifying with a sample. First 10 columns, 3 rows:
-RECORD 0-----------------------------
```

```
 ClientCode             | A180868878
 SnapshotDate           | 2021-01-31
 ActivationDate         | 2018-08-31
 Tenure_Days            | 884
 Last_Trade_Date        | NULL
 Days_Since_Last_Trade  | 885
 Last_Login_Date        | NULL
 Days_Since_Last_Login  | 885
 Last_Deposit_Date      | NULL
 Days_Since_Last_Deposit | 885
-RECORD 1-----------------------------
 ClientCode             | AA02
 SnapshotDate           | 2021-03-31
 ActivationDate         | 2004-03-10
 Tenure_Days            | 6230
 Last_Trade_Date        | NULL
 Days_Since_Last_Trade  | 6231
 Last_Login_Date        | NULL
 Days_Since_Last_Login  | 6231
 Last_Deposit_Date      | NULL
 Days_Since_Last_Deposit | 6231
-RECORD 2-----------------------------
 ClientCode             | AA04
 SnapshotDate           | 2022-04-30
 ActivationDate         | 2004-03-10
 Tenure_Days            | 6625
 Last_Trade_Date        | NULL
 Days_Since_Last_Trade  | 6626
 Last_Login_Date        | NULL
 Days_Since_Last_Login  | 6626
 Last_Deposit_Date      | NULL
 Days_Since_Last_Deposit | 6626
only showing top 3 rows

ABT has 94 columns.

ABT Schema:
root
 |-- ClientCode: string (nullable = true)
 |-- SnapshotDate: date (nullable = true)
 |-- ActivationDate: date (nullable = true)
 |-- Tenure_Days: integer (nullable = true)
 |-- Last_Trade_Date: date (nullable = true)
 |-- Days_Since_Last_Trade: integer (nullable = true)
 |-- Last_Login_Date: date (nullable = true)
 |-- Days_Since_Last_Login: integer (nullable = true)
 |-- Last_Deposit_Date: date (nullable = true)
 |-- Days_Since_Last_Deposit: integer (nullable = true)
 |-- Last_Payout_Date: date (nullable = true)
 |-- Days_Since_Last_Payout: integer (nullable = true)
 |-- Trade_Days_Count_30D: long (nullable = true)
 |-- Trade_Txns_Count_30D: long (nullable = true)
 |-- Trade_Sum_30D: double (nullable = true)
 |-- Trade_Days_Count_90D: long (nullable = true)
```

```
# --- 3. Target Variable Selection and Feature Definition ---

# Select your primary target variable for the first round of modeling
# CHURN_WINDOWS_DAYS was [60, 90, 270, 365]

TARGET_COL = "Is_Churned_Engage_90Days" # Alternative

if TARGET_COL not in abt_df.columns:
    print(f"FATAL ERROR: Target column '{TARGET_COL}' not found in ABT. Available columns: {abt_df.columns}")
    # spark.stop()
    # exit() # Or raise an error
else:
    print(f"Selected Target Variable: {TARGET_COL}")

    # Check target variable distribution
    print(f"\nDistribution of Target Variable ({TARGET_COL}):")
    abt_df.groupBy(TARGET_COL).count().show()

# --- Define Feature Columns ---
# Exclude keys, date columns that are not features, other target labels,
# and descriptive/intermediate columns not intended as direct model inputs.

# Columns to EXCLUDE from features:
# Keys: ClientCode, SnapshotDate, ActivationDate
# Other target labels: Is_Churned_Engage_60Days, _90Days, _270Days, _365Days (excluding the chosen TARGET_COL)
# Date features that are not "Days_Since_Last_": Last_Trade_Date, Last_Login_Date, Last_Deposit_Date, Last_Payout_Date, PreviousMonthEOM
# Descriptive/Qualitative: Payout_Risk_Flag (can be string indexed later if desired as a feature)
# Excel classification (for analysis, not direct model input initially): Historical_Total_Score, Historical_Tag, Excel_Status_Score_S_Dy

excluded_cols_for_features = [
    "ClientCode", "SnapshotDate", "ActivationDate",
    "Last_Trade_Date", "Last_Login_Date", "Last_Deposit_Date", "Last_Payout_Date", "PreviousMonthEOM",
```

```
        "Payout_Risk_Flag", # Can be converted to numeric via StringIndexer if we want to use it
        "Historical_Total_Score", "Historical_Tag", "Excel_Status_Score_S_Dynamic"
    ]

    # Add other churn labels to exclude list (all except the chosen TARGET_COL)
    all_churn_labels = [f"Is_Churned_Engage_{n}Days" for n in [60, 90, 270, 365]]
    for label in all_churn_labels:
        if label != TARGET_COL and label not in excluded_cols_for_features:
            excluded_cols_for_features.append(label)

    # Feature columns are all columns in abt_df MINUS the excluded_cols and the TARGET_COL
    feature_columns = [col_name for col_name in abt_df.columns if col_name not in excluded_cols_for_features and col_name != TARGET_COL]

    print(f"\nNumber of features selected: {len(feature_columns)}")
    print("First 10 feature columns:")
    print(feature_columns[:10])
    print("Last 10 feature columns:")
    print(feature_columns[-10:])

    # Verify all selected feature columns exist and are numeric (or can be treated as such by VectorAssembler)
    # StringIndexer will be needed for any categorical string features if we decide to include them (e.g., Payout_Risk_Flag after indexing)
    # For now, assuming all in 'feature_columns' are numeric or can be cast.
    # We might need to explicitly cast LongType columns to DoubleType for StandardScaler or some models,
    # but VectorAssembler usually handles LongType fine.
```

```
Selected Target Variable: Is_Churned_Engage_90Days

Distribution of Target Variable (Is_Churned_Engage_90Days):
+------------------------+--------+
|Is_Churned_Engage_90Days|   count|
+------------------------+--------+
|                       1|  752632|
|                       0|32444881|
+------------------------+--------+


Number of features selected: 78
First 10 feature columns:
['Tenure_Days', 'Days_Since_Last_Trade', 'Days_Since_Last_Login', 'Days_Since_Last_Deposit', 'Days_Since_Last_Payout', 'Trade_Days_(
Last 10 feature columns:
['Net_Funding_Flow_365D', 'Payout_To_Deposit_Ratio_365D', 'AUM_SnapshotMonth_Monthly', 'AUM_SnapshotMonth_RunningTotal', 'Total_Payo
```

```
# --- 4. Data Splitting (Time-Based, writing to disk) ---

from pyspark.sql.types import DateType # Ensure this import is active
from pyspark.sql.functions import lit, col, min as pyspark_min, max as pyspark_max

model_data_temp_dir = os.path.join(abt_output_dir, "model_data_temp/")
train_df_path = os.path.join(model_data_temp_dir, "train_df.parquet")
test_df_path = os.path.join(model_data_temp_dir, "test_df.parquet")

if not os.path.exists(model_data_temp_dir):
    try:
        os.makedirs(model_data_temp_dir)
        print(f"Created directory: {model_data_temp_dir}")
    except Exception as e_mkdir:
        print(f"Could not create directory {model_data_temp_dir}: {e_mkdir}")
        raise e_mkdir # Stop if we can't create essential directory

if 'abt_df' in locals() and abt_df is not None: # Check if abt_df DataFrame object exists
    print(f"Source ABT for splitting is available (schema has {len(abt_df.columns)} columns).")

    SPLIT_DATE_STR = "2023-03-01"
    split_date_for_print = pd.to_datetime(SPLIT_DATE_STR).date()
    print(f"Splitting data using SnapshotDate. Training data: before {split_date_for_print}")

    # --- Create and Write Training Data ---
    print("\nProcessing Training Data...")
    train_df = abt_df.filter(col("SnapshotDate") < lit(SPLIT_DATE_STR).cast(DateType()))

    train_df.persist() # Persist the (smaller) train_df
    train_count = train_df.count() # Action on train_df
    print(f"Training data: {train_count} rows")

    if train_count > 0:
        print(f"Writing Training data to: {train_df_path}")
        train_df.write.mode("overwrite").parquet(train_df_path)
        print("Training data written successfully.")
        print("SnapshotDate range in Training Data:")
        train_df.select(pyspark_min("SnapshotDate"), pyspark_max("SnapshotDate")).show()
    else:
        print("Training DataFrame is empty. Not writing.")
```

```
    if train_df.is_cached:
        train_df.unpersist()
        print("Unpersisted train_df from memory.")

    # --- Create and Write Test Data ---
    print("\nProcessing Test Data...")
    test_df = abt_df.filter(col("SnapshotDate") >= lit(SPLIT_DATE_STR).cast(DateType()))

    test_df.persist() # Persist the (smaller) test_df
    test_count = test_df.count() # Action on test_df
    print(f"Test data: {test_count} rows")

    if test_count > 0:
        print(f"Writing Test data to: {test_df_path}")
        test_df.write.mode("overwrite").parquet(test_df_path)
        print("Test data written successfully.")
        print("SnapshotDate range in Test Data:")
        test_df.select(pyspark_min("SnapshotDate"), pyspark_max("SnapshotDate")).show()
    else:
        print("Test DataFrame is empty. Not writing.")

    if test_df.is_cached:
        test_df.unpersist()
        print("Unpersisted test_df from memory.")

    # --- Final Verification ---
    source_abt_count_for_pct = train_count + test_count

    if source_abt_count_for_pct > 0 :
        print(f"\nSplit Summary:")
        print(f"  Training data: {train_count} rows ({ (train_count/source_abt_count_for_pct)*100 :.2f}%)")
        print(f"  Test data: {test_count} rows ({ (test_count/source_abt_count_for_pct)*100 :.2f}%)")
    else:
        print("\nSplit Summary: Counts are zero, cannot calculate percentages.")

    if train_count == 0 or test_count == 0:
        print("ERROR: Train or Test DataFrame was empty after processing.")
    else:
        print("Train/Test split and writing to disk successful.")

    # The original abt_df was not persisted, so no need to unpersist it here.
    # If you had other references to it, they would still point to the unmaterialized DF.
    print("Full ABT was not persisted, so no unpersist action needed for it here.")
else:
    print("Skipping Data Splitting as source abt_df is missing or None.")
```

```
Source ABT for splitting is available (schema has 94 columns).
Splitting data using SnapshotDate. Training data: before 2023-03-01

Processing Training Data...
Training data: 30671842 rows
Writing Training data to: /content/drive/MyDrive/Tables/output_abt_final_pred/model_data_temp/train_df.parquet
Training data written successfully.
SnapshotDate range in Training Data:
+-----------------+-----------------+
|min(SnapshotDate)|max(SnapshotDate)|
+-----------------+-----------------+
|       2021-01-31|       2023-02-28|
+-----------------+-----------------+

Unpersisted train_df from memory.

Processing Test Data...
Test data: 2525671 rows
Writing Test data to: /content/drive/MyDrive/Tables/output_abt_final_pred/model_data_temp/test_df.parquet
Test data written successfully.
SnapshotDate range in Test Data:
+-----------------+-----------------+
|min(SnapshotDate)|max(SnapshotDate)|
+-----------------+-----------------+
|       2023-03-31|       2023-04-30|
+-----------------+-----------------+

Unpersisted test_df from memory.

Split Summary:
  Training data: 30671842 rows (92.39%)
  Test data: 2525671 rows (7.61%)
Train/Test split and writing to disk successful.
Full ABT was not persisted, so no unpersist action needed for it here.
```

```
# --- 5. ML Pipeline: Feature Engineering (Assembler, Scaler) & Model Definition ---
print("\n--- Setting up ML Pipeline ---")
```

```
# Define paths (these should be consistent with Cell 4's output paths)
model_data_temp_dir = os.path.join(abt_output_dir, "model_data_temp/")
train_df_path = os.path.join(model_data_temp_dir, "train_df.parquet")
test_df_path = os.path.join(model_data_temp_dir, "test_df.parquet")

try:
    print(f"Loading training data from: {train_df_path}")
    train_df = spark.read.parquet(train_df_path)

    print(f"Loading test data from: {test_df_path}")
    test_df = spark.read.parquet(test_df_path)

    # It's good practice to persist them after loading if they'll be used multiple times
    # by the model training and evaluation stages.
    train_df.persist()
    test_df.persist()

    # Trigger action and verify counts
    train_count_loaded = train_df.count() # Action
    test_count_loaded = test_df.count()   # Action

    print(f"Successfully loaded train_df with {train_count_loaded} rows.")
    print(f"Successfully loaded test_df with {test_count_loaded} rows.")

    if train_count_loaded == 0 or test_count_loaded == 0:
        print("ERROR: Loaded Train or Test DataFrame is empty. Check Parquet files or paths.")
        raise Exception("Empty train or test DataFrame after loading from Parquet.") # Stop execution

    # Ensure TARGET_COL and feature_columns are defined (should be from Cell 3)
    if 'TARGET_COL' not in globals() or 'feature_columns' not in globals():
        print("ERROR: TARGET_COL or feature_columns not defined. Re-run Cell 3.")
        raise Exception("Missing TARGET_COL or feature_columns definition.")

    print(f"Using TARGET_COL: {TARGET_COL}")
    print(f"Number of features to assemble: {len(feature_columns)}")

    # --- Define Pipeline Stages ---
    # Stage 1: VectorAssembler - combines all feature columns into a single vector column
    # Ensure all feature_columns are numeric or will be handled by StringIndexer first if categorical

    # Check for non-numeric types in feature_columns (excluding string features we might index later)
    # For now, assuming all in feature_columns are directly usable by VectorAssembler (numeric)
    # If StringIndexer is used for Payout_Risk_Flag, it would produce an indexed numeric column.

    assembler = VectorAssembler(
        inputCols=feature_columns,
        outputCol="rawFeatures",
        handleInvalid="skip"  # Option to skip rows with nulls in features, or use "keep" / "error"
                              # "keep" would require an Imputer stage before scaler/model
                              # We did fillna(0) for many features in ABT generation.
                              # Days_Since_Last_ features were filled with Tenure+1 or 9999.
    )

    # Stage 2: StandardScaler - scales the feature vector
    scaler = StandardScaler(
        inputCol="rawFeatures",
        outputCol="scaledFeatures",
        withStd=True,
        withMean=True # Centering data by subtracting mean
    )

    # Stage 3: Model - Start with Logistic Regression
    # We'll add weightCol later if needed after checking class imbalance on train_df
    lr = LogisticRegression(
        featuresCol="scaledFeatures",
        labelCol=TARGET_COL # This must be the name of your target column
    )

    # Define the Pipeline
    pipeline = Pipeline(stages=[assembler, scaler, lr])

    print("ML Pipeline defined with Assembler, Scaler, and Logistic Regression.")

except FileNotFoundError:
    print(f"ERROR: Could not read train/test Parquet files from {model_data_temp_dir}. Ensure Cell 4 ran and wrote the files.")
    raise
except Exception as e:
    print(f"Error setting up ML Pipeline: {e}")
    raise
```

```
--- Setting up ML Pipeline ---
Loading training data from: /content/drive/MyDrive/Tables/output_abt_final_pred/model_data_temp/train_df.parquet
Loading test data from: /content/drive/MyDrive/Tables/output_abt_final_pred/model_data_temp/test_df.parquet
Successfully loaded train_df with 30671842 rows.
Successfully loaded test_df with 2525671 rows.
Using TARGET_COL: Is_Churned_Engage_90Days
Number of features to assemble: 78
ML Pipeline defined with Assembler, Scaler, and Logistic Regression.
```

```
# --- 6. Train Logistic Regression Model and Initial Evaluation ---

if 'pipeline' in locals() and 'train_df' in locals() and 'test_df' in locals() \
   and train_df.is_cached and test_df.is_cached: # Check if DFs are loaded and pipeline defined

    print(f"\n--- Training Logistic Regression Model on {TARGET_COL} ---")

    # Before training, let's check class imbalance on the training set for the target column
    print("Class distribution in Training Data:")
    train_df.groupBy(TARGET_COL).count().show()

    # Calculate weights for imbalanced classes if needed (for Logistic Regression)
    # This is a common way to handle imbalance with Spark ML's LogisticRegression
    balance_ratios = train_df.groupBy(TARGET_COL).count().collect()
    count_class_0 = 0
    count_class_1 = 0
    for row in balance_ratios:
        if row[TARGET_COL] == 0:
            count_class_0 = row["count"]
        elif row[TARGET_COL] == 1:
            count_class_1 = row["count"]

    if count_class_0 > 0 and count_class_1 > 0: # Ensure both classes are present
        total_train_count = count_class_0 + count_class_1
        # Weight for class 0: N / (2 * N_0)
        # Weight for class 1: N / (2 * N_1)
        # Spark's weightCol expects a column containing these weights for each instance.
        # Create a 'classWeightCol' in train_df
        balancing_ratio_0 = total_train_count / (2.0 * count_class_0)
        balancing_ratio_1 = total_train_count / (2.0 * count_class_1)

        print(f"Balancing Ratios - Class 0: {balancing_ratio_0:.4f}, Class 1: {balancing_ratio_1:.4f}")

        train_df_weighted = train_df.withColumn(
            "classWeightCol",
            when(col(TARGET_COL) == 0, balancing_ratio_0)
            .otherwise(balancing_ratio_1)
        )
        # Update the Logistic Regression stage in the pipeline to use weightCol
        # Need to access the lr stage (it's the last one in 'pipeline.getStages()')
        lr_model_stage = pipeline.getStages()[-1] # Assuming lr is the last stage
        lr_model_stage.setWeightCol("classWeightCol")
        print("Logistic Regression model in pipeline updated to use 'classWeightCol'.")

        # Fit the pipeline on the weighted training data
        print("Fitting pipeline on weighted training data...")
        pipeline_model = pipeline.fit(train_df_weighted)
    else:
        print("Warning: One or both classes missing in training data, or counts are zero. Training without class weights.")
        # Fit the pipeline on the original training data (no weighting)
        print("Fitting pipeline on training data (no class weighting)...")
        pipeline_model = pipeline.fit(train_df)


    # --- Make Predictions on Test Data ---
    print("\nMaking predictions on test data...")
    predictions_lr = pipeline_model.transform(test_df)

    print("Sample of predictions (showing key columns):")
    predictions_lr.select(TARGET_COL, "rawFeatures", "scaledFeatures", "probability", "prediction").show(5, truncate=50)

    # --- Evaluate Model ---
    print("\nEvaluating Logistic Regression Model...")

    # Evaluator for AUC (Area Under ROC and Area Under PR)
    # Ensure labelCol matches your TARGET_COL and rawPredictionCol is "probability" for AUC
    auc_evaluator = BinaryClassificationEvaluator(labelCol=TARGET_COL, rawPredictionCol="probability", metricName="areaUnderROC")
    roc_auc_lr = auc_evaluator.evaluate(predictions_lr)
    print(f"Area Under ROC (AUC-ROC) for Logistic Regression: {roc_auc_lr:.4f}")

    auc_evaluator.setMetricName("areaUnderPR")
    pr_auc_lr = auc_evaluator.evaluate(predictions_lr)
```

```
    print(f"Area Under PR (AUC-PR) for Logistic Regression: {pr_auc_lr:.4f}")

    # Evaluator for other metrics like Accuracy, Precision, Recall, F1
    # This evaluator uses the 'prediction' column (0s and 1s)
    multi_evaluator = MulticlassClassificationEvaluator(labelCol=TARGET_COL, predictionCol="prediction")

    accuracy_lr = multi_evaluator.setMetricName("accuracy").evaluate(predictions_lr)
    precision_lr = multi_evaluator.setMetricName("weightedPrecision").evaluate(predictions_lr) # Or "precisionByLabel"
    recall_lr = multi_evaluator.setMetricName("weightedRecall").evaluate(predictions_lr)       # Or "recallByLabel"
    f1_lr = multi_evaluator.setMetricName("f1").evaluate(predictions_lr)

    print(f"Accuracy for Logistic Regression: {accuracy_lr:.4f}")
    print(f"Weighted Precision for Logistic Regression: {precision_lr:.4f}")
    print(f"Weighted Recall for Logistic Regression: {recall_lr:.4f}")
    print(f"F1 Score for Logistic Regression: {f1_lr:.4f}")

    # Confusion Matrix (can be calculated manually from predictions)
    print("\nConfusion Matrix for Logistic Regression:")
    predictions_lr.groupBy(TARGET_COL, "prediction").count().orderBy(TARGET_COL, "prediction").show()

    # Unpersist test_df if it won't be immediately reused by another model evaluation in the next cell.
    # Train_df might also be unpersisted if we are done with this model config.
    # For now, keep them persisted as we might try Random Forest next.

else:
    print("Skipping Model Training as pipeline or train/test DataFrames are not defined or cached.")
```

```
--- Training Logistic Regression Model on Is_Churned_Engage_90Days ---
Class distribution in Training Data:
+------------------------+--------+
|Is_Churned_Engage_90Days|   count|
+------------------------+--------+
|                       1|  711395|
|                       0|29960447|
+------------------------+--------+

Balancing Ratios - Class 0: 0.5119, Class 1: 21.5575
Logistic Regression model in pipeline updated to use 'classWeightCol'.
Fitting pipeline on weighted training data...

Making predictions on test data...
Sample of predictions (showing key columns):
+------------------------+--------------------------------------------------+--------------------------------------------------+---
|Is_Churned_Engage_90Days|                                       rawFeatures|                                    scaledFeatures|
+------------------------+--------------------------------------------------+--------------------------------------------------+---
|                       0|(78,[0,1,2,3,4],[6098.0,6099.0,6099.0,6099.0,60...|[1.243709287317651,1.2363978261061948,1.2240129...| [0
|                       0|(78,[0,1,2,3,4,70,71,73],[6043.0,6044.0,6044.0,...|[1.2118857897835587,1.209445359123341,1.1968605...|  [(
|                       0|(78,[0,1,2,3,4],[6025.0,6026.0,6026.0,6026.0,60...|[1.2014708269542194,1.2006245517471341,1.187974...|[0.9
|                       0|(78,[0,1,2,3,4],[6007.0,6008.0,6008.0,6008.0,60...|[1.19105586412488,1.1918037443709275,1.17908807...| [0
|                       0|(78,[0,1,2,3,4],[5959.0,5960.0,5960.0,5960.0,59...|[1.1632826299133088,1.1682815913677096,1.155391...| [0
+------------------------+--------------------------------------------------+--------------------------------------------------+---
only showing top 5 rows


Evaluating Logistic Regression Model...
Area Under ROC (AUC-ROC) for Logistic Regression: 0.9309
Area Under PR (AUC-PR) for Logistic Regression: 0.1399
Accuracy for Logistic Regression: 0.7714
Weighted Precision for Logistic Regression: 0.9840
Weighted Recall for Logistic Regression: 0.7714
F1 Score for Logistic Regression: 0.8564

Confusion Matrix for Logistic Regression:
+------------------------+----------+-------+
|Is_Churned_Engage_90Days|prediction|  count|
+------------------------+----------+-------+
|                       0|       0.0|1908454|
|                       0|       1.0| 575980|
|                       1|       0.0|   1376|
|                       1|       1.0|  39861|
+------------------------+----------+-------+
```

```
# --- 7. Train Random Forest Model and Evaluate ---

# Ensure 'pipeline' (from Cell 5, with LR) is defined
# Ensure 'train_df' and 'test_df' (from Cell 5, loaded from disk) are defined and cached
if 'pipeline' in locals() and \
   'train_df' in locals() and train_df is not None and train_df.is_cached and \
   'test_df' in locals() and test_df is not None and test_df.is_cached:

    print(f"\n--- Training Random Forest Model on {TARGET_COL} ---")

    # Get the original pipeline stages
```

```
# pipeline stages were [assembler, scaler, lr_model_stage]
original_stages = pipeline.getStages()

# Define Random Forest model
# We can also use classWeightCol with RandomForest in Spark ML if we prepare the weights correctly
# However, Random Forests are often inherently better at handling some imbalance due to tree structures.
# Let's try it first without explicit weightCol and see, then consider adding if needed.
# For a more direct comparison, we could use train_df_weighted if it exists from LR step.

rf = RandomForestClassifier(
    featuresCol="scaledFeatures",
    labelCol=TARGET_COL,
    seed=42, # for reproducibility
    numTrees=100 # Default is 20, more trees can improve performance but increase training time
    # maxDepth=5 # Default is 5, can tune this
    # Other hyperparameters can be tuned later: maxBins, minInstancesPerNode, impurity etc.
)

# Create a new pipeline with RandomForest
# Replace the last stage (Logistic Regression) with RandomForest
pipeline_rf = Pipeline(stages=original_stages[:-1] + [rf]) # Keep assembler and scaler

# Determine which training DataFrame to use (weighted or original)
# If class weighting significantly helped LR, it might be beneficial here too.
# However, tree-based models can sometimes handle imbalance naturally or through other params.
# For simplicity, let's use the original train_df for RF first.
# If you used train_df_weighted for LR, that df still has the weightCol. RF can take it.

training_data_for_rf = train_df # Using original non-weighted for now.
# Or, to use weights if you believe they are crucial for RF too:
# if 'train_df_weighted' in locals():
#     training_data_for_rf = train_df_weighted
#     rf.setWeightCol("classWeightCol") # Tell RF to use the weight column
#     print("Random Forest will use 'classWeightCol' from train_df_weighted.")
# else:
#     print("train_df_weighted not found, RF will use original train_df without explicit weights.")

print("Fitting Random Forest pipeline on training data...")
# Ensure training_data_for_rf is cached if it's different from train_df from Cell 4
if not training_data_for_rf.is_cached : training_data_for_rf.persist()

pipeline_model_rf = pipeline_rf.fit(training_data_for_rf)

# --- Make Predictions on Test Data ---
print("\nMaking predictions with Random Forest on test data...")
predictions_rf = pipeline_model_rf.transform(test_df)

print("Sample of Random Forest predictions (showing key columns):")
predictions_rf.select(TARGET_COL, "scaledFeatures", "probability", "prediction").show(5, truncate=50)

# --- Evaluate Random Forest Model ---
print("\nEvaluating Random Forest Model...")

# AUC Evaluator
auc_evaluator_rf = BinaryClassificationEvaluator(labelCol=TARGET_COL, rawPredictionCol="probability", metricName="areaUnderROC")
roc_auc_rf = auc_evaluator_rf.evaluate(predictions_rf)
print(f"Area Under ROC (AUC-ROC) for Random Forest: {roc_auc_rf:.4f}")

auc_evaluator_rf.setMetricName("areaUnderPR")
pr_auc_rf = auc_evaluator_rf.evaluate(predictions_rf)
print(f"Area Under PR (AUC-PR) for Random Forest: {pr_auc_rf:.4f}")

# Multiclass Evaluator for other metrics
multi_evaluator_rf = MulticlassClassificationEvaluator(labelCol=TARGET_COL, predictionCol="prediction")

accuracy_rf = multi_evaluator_rf.setMetricName("accuracy").evaluate(predictions_rf)
precision_rf = multi_evaluator_rf.setMetricName("weightedPrecision").evaluate(predictions_rf)
recall_rf = multi_evaluator_rf.setMetricName("weightedRecall").evaluate(predictions_rf)
f1_rf = multi_evaluator_rf.setMetricName("f1").evaluate(predictions_rf)

print(f"Accuracy for Random Forest: {accuracy_rf:.4f}")
print(f"Weighted Precision for Random Forest: {precision_rf:.4f}")
print(f"Weighted Recall for Random Forest: {recall_rf:.4f}")
print(f"F1 Score for Random Forest: {f1_rf:.4f}")

print("\nConfusion Matrix for Random Forest:")
predictions_rf.groupBy(TARGET_COL, "prediction").count().orderBy(TARGET_COL, "prediction").show()

# Calculate Recall and Precision for the positive class (churners = 1) specifically
tp_rf = predictions_rf.filter((col(TARGET_COL) == 1) & (col("prediction") == 1.0)).count()
fp_rf = predictions_rf.filter((col(TARGET_COL) == 0) & (col("prediction") == 1.0)).count()
fn_rf = predictions_rf.filter((col(TARGET_COL) == 1) & (col("prediction") == 0.0)).count()
```

```
    if (tp_rf + fn_rf) > 0:
        recall_class1_rf = tp_rf / (tp_rf + fn_rf)
        print(f"Recall for Churners (Class 1) - Random Forest: {recall_class1_rf:.4f}")
    else:
        print("No actual churners (Class 1) in test set or no TPs/FNs, cannot calculate specific recall.")

    if (tp_rf + fp_rf) > 0:
        precision_class1_rf = tp_rf / (tp_rf + fp_rf)
        print(f"Precision for Churners (Class 1) - Random Forest: {precision_class1_rf:.4f}")
    else:
        print("No predicted churners (Class 1) by Random Forest, cannot calculate specific precision.")

    # If training_data_for_rf was different and persisted, unpersist it
    if training_data_for_rf is not train_df and training_data_for_rf.is_cached:
        training_data_for_rf.unpersist()
        print("Unpersisted temporary training_data_for_rf.")

else:
    print("Skipping Random Forest Model Training: 'pipeline' from Cell 5, or 'train_df'/'test_df' from Cell 5, are not defined or not ca
    if 'pipeline' not in locals(): print("  Reason: 'pipeline' object not found.")
    if 'train_df' not in locals() or train_df is None : print("  Reason: 'train_df' object not found.")
    elif not train_df.is_cached: print("  Reason: 'train_df' found but not cached.")
    if 'test_df' not in locals() or test_df is None: print("  Reason: 'test_df' object not found.")
    elif not test_df.is_cached: print("  Reason: 'test_df' found but not cached.")
```

```
--- Training Random Forest Model on Is_Churned_Engage_270Days ---
Fitting Random Forest pipeline on training data...

Making predictions with Random Forest on test data...
Sample of Random Forest predictions (showing key columns):
+------------------------+--------------------------------------------------+------------------------------------------+----------
|Is_Churned_Engage_270Days|                                      scaledFeatures|                               probability|prediction
+------------------------+--------------------------------------------------+------------------------------------------+----------
|                       0|[1.243709287317651,1.2363978261061948,1.2240129...| [0.9992029744882909,7.970255117091003E-4]|       0.0
|                       0|[1.2118857897835587,1.209445359123341,1.1968605...|[0.9987996443918539,0.0012003556081460823]|       0.0
|                       0|[1.2014708269542194,1.2006245517471341,1.187974...| [0.9992029744882909,7.970255117091003E-4]|       0.0
|                       0|[1.19105586412488,1.1918037443709275,1.17908807...| [0.9992029744882909,7.970255117091003E-4]|       0.0
|                       0|[1.1632826299133088,1.1682815913677096,1.155391...| [0.9992029744882909,7.970255117091003E-4]|       0.0
+------------------------+--------------------------------------------------+------------------------------------------+----------
only showing top 5 rows


Evaluating Random Forest Model...
Area Under ROC (AUC-ROC) for Random Forest: 0.9876
Area Under PR (AUC-PR) for Random Forest: 0.6750
Accuracy for Random Forest: 0.9779
Weighted Precision for Random Forest: 0.9758
Weighted Recall for Random Forest: 0.9779
F1 Score for Random Forest: 0.9765

Confusion Matrix for Random Forest:
+-------------------------+----------+-------+
|Is_Churned_Engage_270Days|prediction|  count|
+-------------------------+----------+-------+
|                        0|       0.0|2427764|
|                        0|       1.0|  18467|
|                        1|       0.0|  37289|
|                        1|       1.0|  42151|
+-------------------------+----------+-------+

Recall for Churners (Class 1) - Random Forest: 0.5306
Precision for Churners (Class 1) - Random Forest: 0.6954
```

```
# --- 8. Hyperparameter Tuning for Random Forest (Subsampling for Tuning, then Full Train) ---

from pyspark.ml.tuning import TrainValidationSplit
from pyspark.ml.evaluation import BinaryClassificationEvaluator # Ensure evaluator is defined
from pyspark.ml.classification import RandomForestClassifier # Ensure model is defined
from pyspark.ml import Pipeline # Ensure Pipeline is defined

if 'train_df' in locals() and train_df.is_cached and \
   'test_df' in locals() and test_df.is_cached and \
   'pipeline' in locals(): # 'pipeline' from Cell 5 (with LR) is needed for assembler/scaler stages

    print(f"\n--- Hyperparameter Tuning for Random Forest on {TARGET_COL} (Subsampling for Tuning) ---")

    # --- 1. Subsample the training data FOR TUNING ONLY ---
    fraction_for_tuning = 0.05 # Use 5% of the training data for tuning. Adjust as needed.
                               # 5% of 30M is 1.5M rows - still substantial but much less.
                               # If this still OOMs, try 0.01 (1%)

    print(f"Original training data count: {train_df.count()}") # Verify train_df is still there
```

```
sampled_train_df = train_df.sample(withReplacement=False, fraction=fraction_for_tuning, seed=42)
sampled_train_df.persist() # Persist the sample
sampled_train_count = sampled_train_df.count() # Action to materialize
print(f"Tuning on a sample of {sampled_train_count} rows from training data.")

if sampled_train_count == 0:
    print("ERROR: Sampled training data for tuning is empty. Check fraction or original data.")
    # Potentially unpersist and stop
    if sampled_train_df.is_cached: sampled_train_df.unpersist()
    raise Exception("Sampled training data is empty.")

# Get assembler and scaler from the initial pipeline (defined in Cell 5, used by LR)
# This assumes 'pipeline' variable from Cell 5 is still available
try:
    assembler_stage = pipeline.getStages()[0]
    scaler_stage = pipeline.getStages()[1]
except Exception as e_stages:
    print(f"Error getting stages from 'pipeline' (from Cell 5): {e_stages}")
    print("Please ensure Cell 5 was run to define 'pipeline'.")
    if sampled_train_df.is_cached: sampled_train_df.unpersist()
    raise e_stages


rf_for_tuning = RandomForestClassifier(
    featuresCol="scaledFeatures",
    labelCol=TARGET_COL,
    seed=42
)

# Small ParamGrid
param_grid_rf_small = ParamGridBuilder() \
    .addGrid(rf_for_tuning.numTrees, [50, 100]) \
    .addGrid(rf_for_tuning.maxDepth, [5, 10]) \
    .build()
print(f"Number of parameter combinations in SMALLER grid: {len(param_grid_rf_small)}")

tvs_evaluator_rf = BinaryClassificationEvaluator( # Renamed to avoid conflict if cv_evaluator_rf was somehow still in scope
    labelCol=TARGET_COL,
    rawPredictionCol="probability",
    metricName="areaUnderPR"
)

tvs_pipeline_rf = Pipeline(stages=[assembler_stage, scaler_stage, rf_for_tuning])

train_validation_split_rf = TrainValidationSplit(
    estimator=tvs_pipeline_rf,
    estimatorParamMaps=param_grid_rf_small,
    evaluator=tvs_evaluator_rf,
    trainRatio=0.8,
    parallelism=1,
    seed=42
)

print(f"Starting TrainValidationSplit for Random Forest on SAMPLED data...")
tvs_model_rf_on_sample = train_validation_split_rf.fit(sampled_train_df)
print("TrainValidationSplit on SAMPLED data finished.")

best_pipeline_from_sample = tvs_model_rf_on_sample.bestModel
best_rf_model_from_sample = best_pipeline_from_sample.stages[-1] # Get the tuned RFModel

print("\nBest Random Forest Model Parameters found from tuning on SAMPLE:")
best_params_from_sample = {}
param_map_sample = best_rf_model_from_sample.extractParamMap()
for param, value in param_map_sample.items():
    # Check if param is one we tuned or a key RF param
    if hasattr(rf_for_tuning, param.name) and param.name in ["numTrees", "maxDepth", "minInstancesPerNode", "impurity", "featureSubs
         print(f"  {param.name}: {value}")
         best_params_from_sample[param.name] = value

if sampled_train_df.is_cached: # Unpersist the sample
    sampled_train_df.unpersist()
    print("Unpersisted sampled training data.")

# --- 2. Train a FINAL Random Forest model on FULL training data using best parameters ---
print("\n--- Training FINAL Random Forest model on FULL training data with best parameters ---")

final_rf_model = RandomForestClassifier(
    featuresCol="scaledFeatures",
    labelCol=TARGET_COL,
    seed=42
)
# Set the best parameters found from tuning on the sample
```

```
    # Handle cases where a parameter might not have been in our small grid
    if "numTrees" in best_params_from_sample: final_rf_model.setNumTrees(best_params_from_sample["numTrees"])
    else: final_rf_model.setNumTrees(100) # Fallback default

    if "maxDepth" in best_params_from_sample: final_rf_model.setMaxDepth(best_params_from_sample["maxDepth"])
    else: final_rf_model.setMaxDepth(5) # Fallback default

    if "minInstancesPerNode" in best_params_from_sample: final_rf_model.setMinInstancesPerNode(best_params_from_sample["minInstancesPerN
    # else default is 1

    if "impurity" in best_params_from_sample: final_rf_model.setImpurity(best_params_from_sample["impurity"])
    # else default is "gini"

    # Optional: Use class weights if LR results or initial RF showed it's critical
    # if 'train_df_weighted' in locals():
    #     print("Applying class weights to final Random Forest model.")
    #     final_rf_model.setWeightCol("classWeightCol")
    #     training_data_for_final_rf = train_df_weighted
    # else:
    #     training_data_for_final_rf = train_df
    training_data_for_final_rf = train_df # Using original train_df for now

    final_pipeline_rf = Pipeline(stages=[assembler_stage, scaler_stage, final_rf_model])

    print("Fitting FINAL Random Forest pipeline on FULL training data...")
    final_pipeline_model_rf_tuned = final_pipeline_rf.fit(training_data_for_final_rf)
    print("FINAL Random Forest model training finished.")

    # --- 3. Evaluate the FINAL Tuned Random Forest Model ---
    print("\nMaking predictions with the FINAL Tuned Random Forest on test data...")
    final_predictions_rf_tuned = final_pipeline_model_rf_tuned.transform(test_df)

    print("Sample of FINAL Tuned Random Forest predictions:")
    final_predictions_rf_tuned.select(TARGET_COL, "scaledFeatures", "probability", "prediction").show(5, truncate=50)

    print("\nEvaluating FINAL Tuned Random Forest Model...")
    # (Evaluation metric calculations - AUC-ROC, AUC-PR, Accuracy, etc. - are the same as before, just use final_predictions_rf_tuned)
    evaluator_final_rf = BinaryClassificationEvaluator(labelCol=TARGET_COL, rawPredictionCol="probability")
    roc_auc_final_rf = evaluator_final_rf.setMetricName("areaUnderROC").evaluate(final_predictions_rf_tuned)
    pr_auc_final_rf = evaluator_final_rf.setMetricName("areaUnderPR").evaluate(final_predictions_rf_tuned)
    print(f"Area Under ROC (AUC-ROC) for FINAL Tuned RF: {roc_auc_final_rf:.4f}")
    print(f"Area Under PR (AUC-PR) for FINAL Tuned RF: {pr_auc_final_rf:.4f}")

    multi_eval_final_rf = MulticlassClassificationEvaluator(labelCol=TARGET_COL, predictionCol="prediction")
    # ... (accuracy, precision, recall, f1, confusion matrix, class 1 metrics - same as before) ...
    accuracy_final_rf = multi_eval_final_rf.setMetricName("accuracy").evaluate(final_predictions_rf_tuned)
    precision_final_rf = multi_eval_final_rf.setMetricName("weightedPrecision").evaluate(final_predictions_rf_tuned)
    recall_final_rf = multi_eval_final_rf.setMetricName("weightedRecall").evaluate(final_predictions_rf_tuned)
    f1_final_rf = multi_eval_final_rf.setMetricName("f1").evaluate(final_predictions_rf_tuned)
    print(f"Accuracy for FINAL Tuned RF: {accuracy_final_rf:.4f}")
    print(f"Weighted Precision for FINAL Tuned RF: {precision_final_rf:.4f}")
    print(f"Weighted Recall for FINAL Tuned RF: {recall_final_rf:.4f}")
    print(f"F1 Score for FINAL Tuned RF: {f1_final_rf:.4f}")

    print("\nConfusion Matrix for FINAL Tuned Random Forest:")
    final_predictions_rf_tuned.groupBy(TARGET_COL, "prediction").count().orderBy(TARGET_COL, "prediction").show()

    tp_final_rf = final_predictions_rf_tuned.filter((col(TARGET_COL) == 1) & (col("prediction") == 1.0)).count()
    fp_final_rf = final_predictions_rf_tuned.filter((col(TARGET_COL) == 0) & (col("prediction") == 1.0)).count()
    fn_final_rf = final_predictions_rf_tuned.filter((col(TARGET_COL) == 1) & (col("prediction") == 0.0)).count()
    if (tp_final_rf + fn_final_rf) > 0: recall_class1_final_rf = tp_final_rf / (tp_final_rf + fn_final_rf); print(f"Recall for Churners
    else: print("Cannot calculate specific recall for Class 1 (FINAL Tuned RF).")
    if (tp_final_rf + fp_final_rf) > 0: precision_class1_final_rf = tp_final_rf / (tp_final_rf + fp_final_rf); print(f"Precision for Chu
    else: print("Cannot calculate specific precision for Class 1 (FINAL Tuned RF).")

else:
    print("Skipping Random Forest Hyperparameter Tuning as 'train_df', 'test_df', or 'pipeline' (from Cell 5) is not available or cached
```

```
--- Hyperparameter Tuning for Random Forest on Is_Churned_Engage_270Days (Subsampling for Tuning) ---
Original training data count: 30671842
Tuning on a sample of 1532235 rows from training data.
Number of parameter combinations in SMALLER grid: 4
Starting TrainValidationSplit for Random Forest on SAMPLED data...
TrainValidationSplit on SAMPLED data finished.

Best Random Forest Model Parameters found from tuning on SAMPLE:
  featureSubsetStrategy: auto
  impurity: gini
  maxDepth: 10
  minInstancesPerNode: 1
  numTrees: 50
Unpersisted sampled training data.
```

```
--- Training FINAL Random Forest model on FULL training data with best parameters ---
Fitting FINAL Random Forest pipeline on FULL training data...
FINAL Random Forest model training finished.

Making predictions with the FINAL Tuned Random Forest on test data...
Sample of FINAL Tuned Random Forest predictions:
+------------------------+--------------------------------------------------+-----------+----------+
|Is_Churned_Engage_270Days|                                    scaledFeatures|probability|prediction|
+------------------------+--------------------------------------------------+-----------+----------+
|                       0|[1.243709287317651,1.2363978261061948,1.2240129...|  [1.0,0.0]|       0.0|
|                       0|[1.2118857897835587,1.209445359123341,1.1968605...|  [1.0,0.0]|       0.0|
|                       0|[1.2014708269542194,1.2006245517471341,1.187974...|  [1.0,0.0]|       0.0|
|                       0|[1.19105586412488,1.1918037443709275,1.17908807...|  [1.0,0.0]|       0.0|
|                       0|[1.1632826299133088,1.1682815913677096,1.155391...|  [1.0,0.0]|       0.0|
+------------------------+--------------------------------------------------+-----------+----------+
only showing top 5 rows


Evaluating FINAL Tuned Random Forest Model...
Area Under ROC (AUC-ROC) for FINAL Tuned RF: 0.9893
Area Under PR (AUC-PR) for FINAL Tuned RF: 0.7230
Accuracy for FINAL Tuned RF: 0.9782
Weighted Precision for FINAL Tuned RF: 0.9795
Weighted Recall for FINAL Tuned RF: 0.9782
F1 Score for FINAL Tuned RF: 0.9788

Confusion Matrix for FINAL Tuned Random Forest:
+------------------------+----------+-------+
|Is_Churned_Engage_270Days|prediction|  count|
+------------------------+----------+-------+
|                       0|       0.0|2414254|
|                       0|       1.0|  31977|
|                       1|       0.0|  22957|
|                       1|       1.0|  56483|
+------------------------+----------+-------+

Recall for Churners (Class 1) - FINAL Tuned RF: 0.7110
Precision for Churners (Class 1) - FINAL Tuned RF: 0.6385
```

```python
# --- 8. Re-train FINAL Best RF Model (with known best params), Evaluate, & SAVE ---

from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator # Ensure evaluators are in scope
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml import Pipeline
from pyspark.sql.functions import col # Ensure col is in scope

# Ensure train_df, test_df, pipeline (for assembler/scaler), TARGET_COL, feature_columns are available
# These should have been set up by running Cells 1, (part of 2), 3, 4, 5 in this resumed session.
if 'train_df' in locals() and train_df.is_cached and \
   'test_df' in locals() and test_df.is_cached and \
   'pipeline' in locals() and 'TARGET_COL' in globals() and 'feature_columns' in globals():

    print(f"\n--- Re-training FINAL Best Random Forest on {TARGET_COL} with Known Best Parameters ---")

    try:
        # Get assembler and scaler stages from the 'pipeline' object defined in Cell 5
        assembler_stage = pipeline.getStages()[0]
        scaler_stage = pipeline.getStages()[1]
    except Exception as e_stages:
        print(f"Error getting assembler/scaler stages from 'pipeline' (from Cell 5): {e_stages}")
        print("Please ensure Cell 5 was run to define 'pipeline', and Cells 1-4 for its inputs.")
        raise e_stages

    # --- Use the BEST parameters from your PREVIOUS successful Cell 8 run ---
    print("Using PREVIOUSLY DETERMINED Best Random Forest Model Parameters:")
    best_numTrees = 50
    best_maxDepth = 10
    best_minInstancesPerNode = 1
    best_impurity = "gini"
    # For featureSubsetStrategy, 'auto' is fine, or be explicit e.g. 'sqrt' which is common for classification
    # If 'auto' was in output, it means Spark picked. Let's let it pick or set a common default.
    best_featureSubsetStrategy = "auto" # Or "sqrt", "log2", "onethird"

    print(f"  numTrees: {best_numTrees}")
    print(f"  maxDepth: {best_maxDepth}")
    print(f"  minInstancesPerNode: {best_minInstancesPerNode}")
    print(f"  impurity: {best_impurity}")
    print(f"  featureSubsetStrategy: {best_featureSubsetStrategy}") # Default is often 'auto' which maps to sqrt for classification

    final_rf_model = RandomForestClassifier(
        featuresCol="scaledFeatures",
        labelCol=TARGET_COL,
        seed=42, # Keep seed for reproducibility of this specific model build
        numTrees=best_numTrees,
        maxDepth=best_maxDepth,
```

```
            minInstancesPerNode=best_minInstancesPerNode,
            impurity=best_impurity,
            featureSubsetStrategy=best_featureSubsetStrategy
        )

        training_data_for_final_rf = train_df # train_df was loaded and persisted in Cell 5

        final_pipeline_rf_obj = Pipeline(stages=[assembler_stage, scaler_stage, final_rf_model])

        print("\nFitting FINAL Random Forest pipeline on FULL training data...")
        # This is the variable that holds the final trained pipeline model
        final_pipeline_model_rf_tuned = final_pipeline_rf_obj.fit(training_data_for_final_rf)
        print("FINAL Random Forest model training finished.")

        # --- Evaluate the FINAL (Re-trained) Random Forest Model ---
        print("\nMaking predictions with the FINAL (Re-trained) Random Forest on test data...")

        predictions_base_df = final_pipeline_model_rf_tuned.transform(test_df)
        # Select only necessary columns for evaluation IMMEDIATELY to save memory
        final_predictions_rf_tuned = predictions_base_df.select(
            TARGET_COL,
            "probability",
            "prediction"
        )
        final_predictions_rf_tuned.persist()
        eval_predictions_count = final_predictions_rf_tuned.count()
        print(f"Predictions for evaluation generated and persisted ({eval_predictions_count} rows).")

        print("Sample of FINAL Random Forest predictions:")
        final_predictions_rf_tuned.show(5, truncate=50)


        print("\nEvaluating FINAL (Re-trained) Random Forest Model...")
        evaluator_final_rf = BinaryClassificationEvaluator(labelCol=TARGET_COL, rawPredictionCol="probability")
        # These variables will be recreated and will be available for Cell 10 (Save Metrics)
        roc_auc_final_rf = evaluator_final_rf.setMetricName("areaUnderROC").evaluate(final_predictions_rf_tuned)
        pr_auc_final_rf = evaluator_final_rf.setMetricName("areaUnderPR").evaluate(final_predictions_rf_tuned)
        print(f"Area Under ROC (AUC-ROC) for FINAL RF: {roc_auc_final_rf:.4f}")
        print(f"Area Under PR (AUC-PR) for FINAL RF: {pr_auc_final_rf:.4f}")

        multi_eval_final_rf = MulticlassClassificationEvaluator(labelCol=TARGET_COL, predictionCol="prediction")
        accuracy_final_rf = multi_eval_final_rf.setMetricName("accuracy").evaluate(final_predictions_rf_tuned)
        precision_final_rf_weighted = multi_eval_final_rf.setMetricName("weightedPrecision").evaluate(final_predictions_rf_tuned) # Renamed
        recall_final_rf_weighted = multi_eval_final_rf.setMetricName("weightedRecall").evaluate(final_predictions_rf_tuned) # Renamed
        f1_final_rf = multi_eval_final_rf.setMetricName("f1").evaluate(final_predictions_rf_tuned) # Weighted F1
        print(f"Accuracy for FINAL RF: {accuracy_final_rf:.4f}")
        print(f"Weighted Precision for FINAL RF: {precision_final_rf_weighted:.4f}")
        print(f"Weighted Recall for FINAL RF: {recall_final_rf_weighted:.4f}")
        print(f"F1 Score for FINAL RF: {f1_final_rf:.4f}")

        print("\nConfusion Matrix for FINAL Random Forest:")
        final_predictions_rf_tuned.groupBy(TARGET_COL, "prediction").count().orderBy(TARGET_COL, "prediction").show()

        tp_final_rf = final_predictions_rf_tuned.filter((col(TARGET_COL) == 1) & (col("prediction") == 1.0)).count()
        fp_final_rf = final_predictions_rf_tuned.filter((col(TARGET_COL) == 0) & (col("prediction") == 1.0)).count()
        fn_final_rf = final_predictions_rf_tuned.filter((col(TARGET_COL) == 1) & (col("prediction") == 0.0)).count()

        # These specific class 1 metrics will be available for Cell 10
        recall_class1_final_rf = tp_final_rf / (tp_final_rf + fn_final_rf) if (tp_final_rf + fn_final_rf) > 0 else 0.0
        precision_class1_final_rf = tp_final_rf / (tp_final_rf + fp_final_rf) if (tp_final_rf + fp_final_rf) > 0 else 0.0
        print(f"Recall for Churners (Class 1) - FINAL RF: {recall_class1_final_rf:.4f}")
        print(f"Precision for Churners (Class 1) - FINAL RF: {precision_class1_final_rf:.4f}")

        # --- SAVE THE FINAL PIPELINE MODEL ---
        # This is now part of this modified Cell 8
        if 'final_pipeline_model_rf_tuned' in locals():
            best_rf_model_save_path = os.path.join(abt_output_dir, f"best_rf_pipeline_model_{TARGET_COL}")
            try:
                print(f"\nSaving FINAL Random Forest pipeline model to: {best_rf_model_save_path}")
                final_pipeline_model_rf_tuned.write().overwrite().save(best_rf_model_save_path)
                print(f"FINAL Random Forest pipeline model saved successfully.")
            except Exception as e_save:
                print(f"ERROR saving final RF model: {e_save}")

        if final_predictions_rf_tuned.is_cached:
            final_predictions_rf_tuned.unpersist()
            print("Unpersisted final_predictions_rf_tuned.")
    else:
        print("Skipping FINAL Random Forest Training as 'train_df', 'test_df', 'pipeline', 'TARGET_COL', or 'feature_columns' is not availab
```

```
--- Re-training FINAL Best Random Forest on Is_Churned_Engage_90Days with Known Best Parameters ---
```

```
Using PREVIOUSLY DETERMINED Best Random Forest Model Parameters:
  numTrees: 50
  maxDepth: 10
  minInstancesPerNode: 1
  impurity: gini
  featureSubsetStrategy: auto

Fitting FINAL Random Forest pipeline on FULL training data...
FINAL Random Forest model training finished.

Making predictions with the FINAL (Re-trained) Random Forest on test data...
Predictions for evaluation generated and persisted (2525671 rows).
Sample of FINAL Random Forest predictions:
+-----------------------+-----------+----------+
|Is_Churned_Engage_90Days|probability|prediction|
+-----------------------+-----------+----------+
|                      0|  [1.0,0.0]|       0.0|
|                      0|  [1.0,0.0]|       0.0|
|                      0|  [1.0,0.0]|       0.0|
|                      0|  [1.0,0.0]|       0.0|
|                      0|  [1.0,0.0]|       0.0|
+-----------------------+-----------+----------+
only showing top 5 rows


Evaluating FINAL (Re-trained) Random Forest Model...
Area Under ROC (AUC-ROC) for FINAL RF: 0.9914
Area Under PR (AUC-PR) for FINAL RF: 0.6077
Accuracy for FINAL RF: 0.9868
Weighted Precision for FINAL RF: 0.9851
Weighted Recall for FINAL RF: 0.9868
F1 Score for FINAL RF: 0.9857

Confusion Matrix for FINAL Random Forest:
+-----------------------+----------+-------+
|Is_Churned_Engage_90Days|prediction|  count|
+-----------------------+----------+-------+
|                      0|       0.0|2473811|
|                      0|       1.0|  10623|
|                      1|       0.0|  22650|
|                      1|       1.0|  18587|
+-----------------------+----------+-------+

Recall for Churners (Class 1) - FINAL RF: 0.4507
Precision for Churners (Class 1) - FINAL RF: 0.6363

Saving FINAL Random Forest pipeline model to: /content/drive/MyDrive/Tables/output_abt_final_pred/best_rf_pipeline_model_Is_Churned_
FINAL Random Forest pipeline model saved successfully.
Unpersisted final_predictions_rf_tuned.
```

```python
# --- 9. Train Gradient-Boosted Trees (GBT) Model and Evaluate (Reduced Complexity) ---

# Ensure train_df, test_df, pipeline (for assembler/scaler), TARGET_COL, feature_columns are available
if 'train_df' in locals() and train_df.is_cached and \
   'test_df' in locals() and test_df.is_cached and \
   'pipeline' in locals(): # 'pipeline' from Cell 5 (with LR) used to get assembler/scaler stages

    print(f"\n--- Training Gradient-Boosted Trees (GBT) Model on {TARGET_COL} (Reduced Complexity) ---")

    # Get assembler and scaler from the initial pipeline (defined in Cell 5)
    try:
        assembler_stage_gbt = pipeline.getStages()[0]
        scaler_stage_gbt = pipeline.getStages()[1]
    except Exception as e_stages_gbt:
        print(f"Error getting assembler/scaler stages from 'pipeline' (from Cell 5): {e_stages_gbt}")
        print("Please ensure Cell 5 was run to define 'pipeline'.")
        raise e_stages_gbt

    # Define GBT model with REDUCED parameters
    gbt = GBTClassifier(
        featuresCol="scaledFeatures",
        labelCol=TARGET_COL,
        seed=42,
        maxIter=20,  # REDUCED: Number of trees (iterations). Was 50. Try 20 or even 10.
        maxDepth=4,   # REDUCED: Default is 5. Try 4 or even 3.
        stepSize=0.1  # Learning rate. Default is 0.1. Keep for now.
        # subsamplingRate=0.8 # Optionally add subsampling if still OOMing
    )
    print(f"GBT Parameters: maxIter={gbt.getMaxIter()}, maxDepth={gbt.getMaxDepth()}, stepSize={gbt.getStepSize()}")

    # Create a new pipeline with GBT
    pipeline_gbt = Pipeline(stages=[assembler_stage_gbt, scaler_stage_gbt, gbt])

    # Training data for GBT: Using the original train_df
    training_data_for_gbt = train_df
```

```
    # Ensure train_df is persisted (should be from Cell 5)
    if not training_data_for_gbt.is_cached :
        print("Persisting training_data_for_gbt for GBT fitting...")
        training_data_for_gbt.persist()

    print("Fitting GBT pipeline on training data...")
    try:
        pipeline_model_gbt = pipeline_gbt.fit(training_data_for_gbt)
        print("GBT pipeline fitting finished.")
    except Exception as e_fit:
        print(f"ERROR during GBT pipeline_gbt.fit(): {e_fit}")
        # If fit fails, we should not proceed to transform and evaluate
        raise e_fit


    # --- Make Predictions on Test Data ---
    print("\nMaking predictions with GBT on test data...")
    predictions_gbt = pipeline_model_gbt.transform(test_df)

    print("Sample of GBT predictions (showing key columns):")
    predictions_gbt.select(TARGET_COL, "scaledFeatures", "probability", "prediction").show(5, truncate=50)

    # --- Evaluate GBT Model ---
    print("\nEvaluating GBT Model...")

    # AUC Evaluator
    auc_evaluator_gbt = BinaryClassificationEvaluator(labelCol=TARGET_COL, rawPredictionCol="probability", metricName="areaUnderROC")
    roc_auc_gbt = auc_evaluator_gbt.evaluate(predictions_gbt)
    print(f"Area Under ROC (AUC-ROC) for GBT: {roc_auc_gbt:.4f}")

    auc_evaluator_gbt.setMetricName("areaUnderPR")
    pr_auc_gbt = auc_evaluator_gbt.evaluate(predictions_gbt)
    print(f"Area Under PR (AUC-PR) for GBT: {pr_auc_gbt:.4f}")

    # Multiclass Evaluator for other metrics
    multi_evaluator_gbt = MulticlassClassificationEvaluator(labelCol=TARGET_COL, predictionCol="prediction")

    accuracy_gbt = multi_evaluator_gbt.setMetricName("accuracy").evaluate(predictions_gbt)
    precision_gbt = multi_evaluator_gbt.setMetricName("weightedPrecision").evaluate(predictions_gbt)
    recall_gbt = multi_evaluator_gbt.setMetricName("weightedRecall").evaluate(predictions_gbt)
    f1_gbt = multi_evaluator_gbt.setMetricName("f1").evaluate(predictions_gbt)

    print(f"Accuracy for GBT: {accuracy_gbt:.4f}")
    print(f"Weighted Precision for GBT: {precision_gbt:.4f}")
    print(f"Weighted Recall for GBT: {recall_gbt:.4f}")
    print(f"F1 Score for GBT: {f1_gbt:.4f}")

    print("\nConfusion Matrix for GBT:")
    predictions_gbt.groupBy(TARGET_COL, "prediction").count().orderBy(TARGET_COL, "prediction").show()

    # Calculate Recall and Precision for the positive class (churners = 1) specifically
    tp_gbt = predictions_gbt.filter((col(TARGET_COL) == 1) & (col("prediction") == 1.0)).count()
    fp_gbt = predictions_gbt.filter((col(TARGET_COL) == 0) & (col("prediction") == 1.0)).count()
    fn_gbt = predictions_gbt.filter((col(TARGET_COL) == 1) & (col("prediction") == 0.0)).count()

    if (tp_gbt + fn_gbt) > 0:
        recall_class1_gbt = tp_gbt / (tp_gbt + fn_gbt)
        print(f"Recall for Churners (Class 1) - GBT: {recall_class1_gbt:.4f}")
    else:
        print("No actual churners (Class 1) in test set or no TPs/FNs for GBT, cannot calculate specific recall.")

    if (tp_gbt + fp_gbt) > 0:
        precision_class1_gbt = tp_gbt / (tp_gbt + fp_gbt)
        print(f"Precision for Churners (Class 1) - GBT: {precision_class1_gbt:.4f}")
    else:
        print("No predicted churners (Class 1) by GBT, cannot calculate specific precision.")

else:
    print("Skipping GBT Model Training as 'pipeline' (from Cell 5) or 'train_df'/'test_df' are not available/cached.")
```

```
 --- Training Gradient-Boosted Trees (GBT) Model on Is_Churned_Engage_270Days (Reduced Complexity) ---
 GBT Parameters: maxIter=20, maxDepth=4, stepSize=0.1
 Fitting GBT pipeline on training data...
```

```
# --- 10. Save Best Tuned Random Forest Model & Key Metrics ---

# Ensure the best model object and its metrics from Cell 8 are in scope.
# If you restarted the kernel after Cell 8, you'd need to reload the saved model first if you saved it,
# or re-run Cell 8 (which is long). For now, assuming Cell 8 just completed and variables are available.

if 'final_pipeline_model_rf_tuned' in locals() and \
```

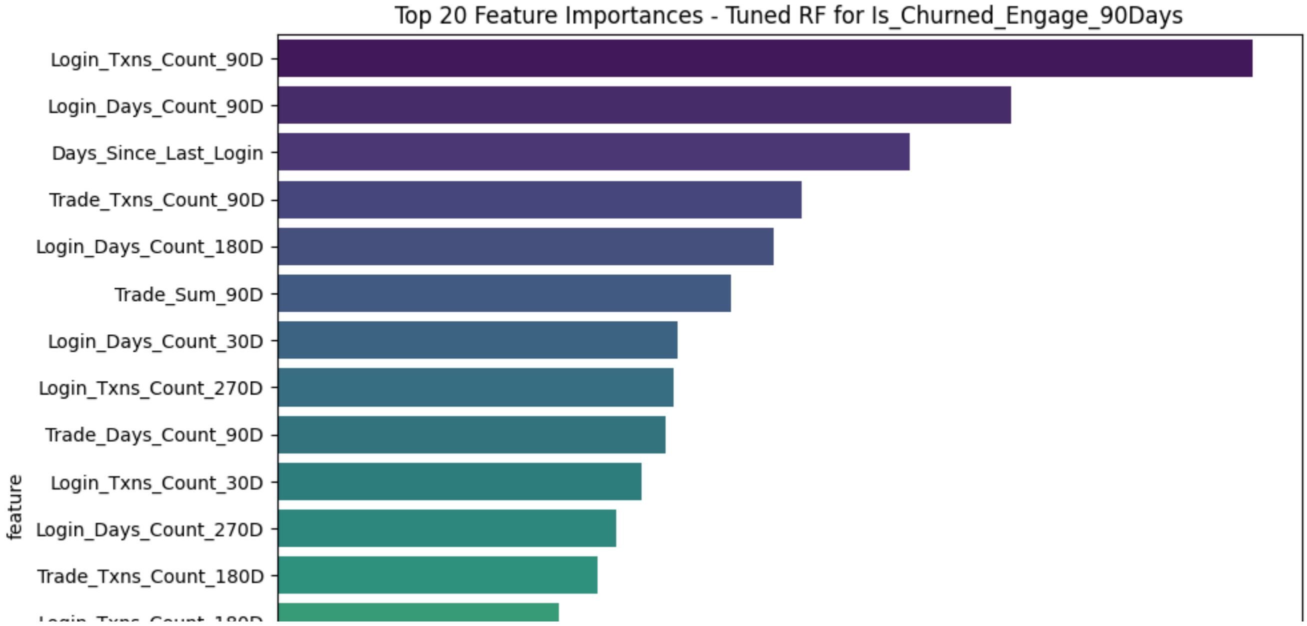
```
    'roc_auc_final_rf' in locals() and 'pr_auc_final_rf' in locals() and \
    'accuracy_final_rf' in locals() and 'f1_final_rf' in locals() and \
    'recall_class1_final_rf' in locals() and 'precision_class1_final_rf' in locals() and \
    'TARGET_COL' in globals():

    print(f"\n--- Saving Best Tuned Random Forest Model and Metrics for Target: {TARGET_COL} ---")

    # 1. Save the PipelineModel
    best_rf_model_save_path = os.path.join(abt_output_dir, f"best_rf_pipeline_model_{TARGET_COL}") # Include target in name
    try:
        final_pipeline_model_rf_tuned.write().overwrite().save(best_rf_model_save_path)
        print(f"Best Tuned Random Forest pipeline model saved to: {best_rf_model_save_path}")
    except Exception as e_save:
        print(f"Error saving best RF model: {e_save}")

    # 2. Save the Metrics (e.g., to a text file or a structured file like JSON/CSV)
    metrics_summary_rf = {
        "target_variable": TARGET_COL,
        "model_type": "RandomForest_Tuned",
        "AUC_ROC": roc_auc_final_rf,
        "AUC_PR": pr_auc_final_rf,
        "Accuracy": accuracy_final_rf,
        "Weighted_F1": f1_final_rf,
        "Recall_Class1_Churners": recall_class1_final_rf,
        "Precision_Class1_Churners": precision_class1_final_rf,
        "Parameters": {} # Populate with best_params_from_sample if available
    }

    # Populate parameters if best_params_from_sample was created and available from Cell 8
    if 'best_params_from_sample' in globals(): # best_params_from_sample was from tuning on sample
        metrics_summary_rf["Parameters"] = best_params_from_sample
    elif 'best_rf_model_params' in locals(): # If best_rf_model_params (from full model) was defined
         # Extract params from the actual final_rf_model stage if best_params_from_sample isn't available
        try:
            final_rf_stage_in_pipeline = final_pipeline_model_rf_tuned.stages[-1]
            param_map = final_rf_stage_in_pipeline.extractParamMap()
            extracted_params = {}
            for param, value in param_map.items():
                if hasattr(final_rf_stage_in_pipeline, param.name) and param.name in ["numTrees", "maxDepth", "minInstancesPerNode", "ir
                    extracted_params[param.name] = value
            metrics_summary_rf["Parameters"] = extracted_params
        except Exception as e_param_extract:
            print(f"Could not extract parameters from final tuned model: {e_param_extract}")


    metrics_file_path = os.path.join(abt_output_dir, f"metrics_summary_rf_{TARGET_COL}.json")
    try:
        import json
        with open(metrics_file_path, 'w') as f:
            json.dump(metrics_summary_rf, f, indent=4)
        print(f"Metrics summary saved to: {metrics_file_path}")
    except Exception as e_metrics_save:
        print(f"Error saving metrics summary: {e_metrics_save}")

else:
    print("Skipping saving model/metrics: Required variables from Cell 8 (RF Tuning) not found.")
```

```
--- Saving Best Tuned Random Forest Model and Metrics for Target: Is_Churned_Engage_90Days ---
Best Tuned Random Forest pipeline model saved to: /content/drive/MyDrive/Tables/output_abt_final_pred/best_rf_pipeline_model_Is_Chur
Metrics summary saved to: /content/drive/MyDrive/Tables/output_abt_final_pred/metrics_summary_rf_Is_Churned_Engage_90Days.json
```

```
# --- 11. Analyze Feature Importances from Best Tuned Random Forest ---

if 'final_pipeline_model_rf_tuned' in locals() and 'feature_columns' in globals():
    print(f"\n--- Feature Importances from Best Tuned Random Forest for Target: {TARGET_COL} ---")

    try:
        # The Random Forest model is the last stage in the best_pipeline_model_rf
        rf_model_stage_from_tuned_pipeline = final_pipeline_model_rf_tuned.stages[-1]

        if isinstance(rf_model_stage_from_tuned_pipeline, RandomForestClassifier) or \
           hasattr(rf_model_stage_from_tuned_pipeline, 'featureImportances'): # Check if it's indeed RF model

            importances = rf_model_stage_from_tuned_pipeline.featureImportances

            # feature_columns should be defined in Cell 3 and used by VectorAssembler
            # The assembler was the first stage of the pipeline
            # assembler_stage = final_pipeline_model_rf_tuned.stages[0]
            # feature_columns_from_assembler = assembler_stage.getInputCols()
            # Using the 'feature_columns' variable directly from Cell 3 is usually fine if consistent.
```

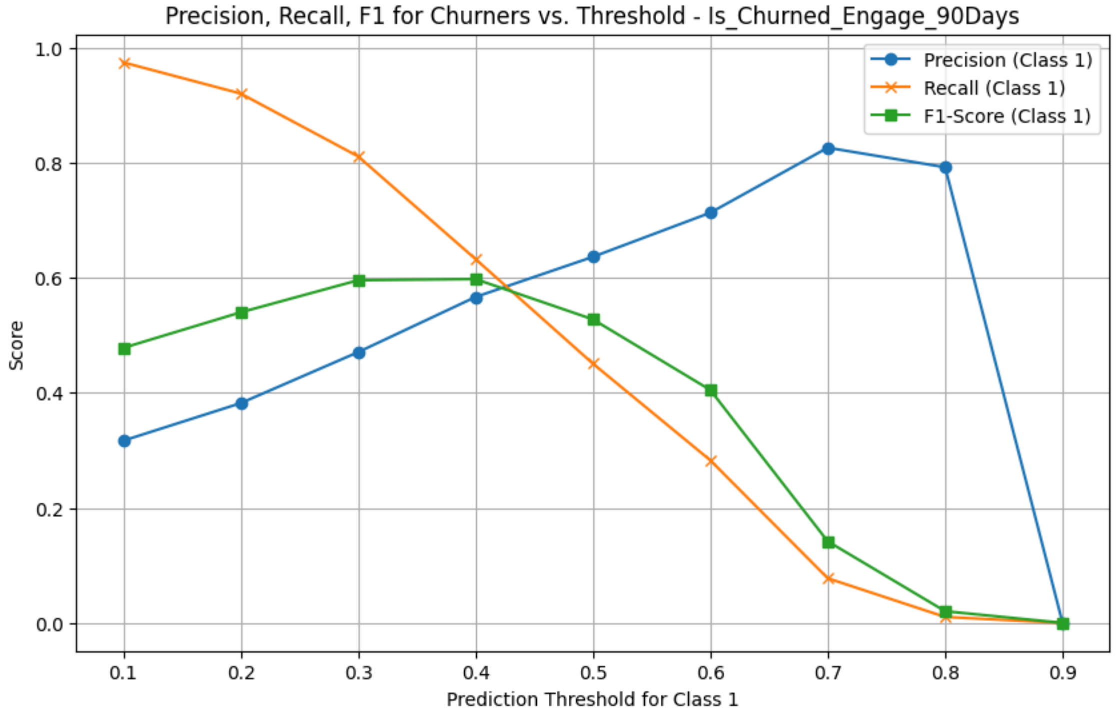
```
            if len(feature_columns) == len(importances):
                feature_importances_pd = pd.DataFrame({
                    'feature': feature_columns,
                    'importance': importances.toArray()
                }).sort_values('importance', ascending=False)

                print("\nTop 20 Feature Importances:")
                print(feature_importances_pd.head(20))

                # Plot feature importances
                plt.figure(figsize=(10, 8))
                top_n = 20
                sns.barplot(x='importance', y='feature', data=feature_importances_pd.head(top_n), palette="viridis")
                plt.title(f'Top {top_n} Feature Importances - Tuned RF for {TARGET_COL}')
                plt.tight_layout()
                plt.show()
            else:
                print(f"Error: Length of feature_columns ({len(feature_columns)}) does not match length of importances ({len(importances
                print(f"Feature columns from assembler: {final_pipeline_model_rf_tuned.stages[0].getInputCols()}")


        else:
            print("The last stage of the best pipeline model is not a RandomForestClassifier model or has no featureImportances attribut

    except Exception as e_fi:
        print(f"Error getting or plotting feature importances: {e_fi}")
else:
    print("Skipping Feature Importance Analysis: Best tuned RF model or feature_columns not found.")
```

```
--- Feature Importances from Best Tuned Random Forest for Target: Is_Churned_Engage_90Days ---

Top 20 Feature Importances:
                  feature  importance
23    Login_Txns_Count_90D    0.106688
22    Login_Days_Count_90D    0.080181
2    Days_Since_Last_Login    0.069171
9     Trade_Txns_Count_90D    0.057278
24   Login_Days_Count_180D    0.054245
10           Trade_Sum_90D    0.049626
20    Login_Days_Count_30D    0.043799
27   Login_Txns_Count_270D    0.043380
8     Trade_Days_Count_90D    0.042418
21    Login_Txns_Count_30D    0.039893
26   Login_Days_Count_270D    0.037111
12   Trade_Txns_Count_180D    0.034968
25   Login_Txns_Count_180D    0.030861
15   Trade_Txns_Count_270D    0.027927
17   Trade_Days_Count_365D    0.026167
13          Trade_Sum_180D    0.023114
11   Trade_Days_Count_180D    0.021898
14   Trade_Days_Count_270D    0.020526
28   Login_Days_Count_365D    0.017909
<ipython-input-8-1bdc060e29fa>:33: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `

  sns.barplot(x='importance', y='feature', data=feature_importances_pd.head(top_n), palette="viridis")
```



```
# --- 12. Adjust Prediction Threshold for Tuned Random Forest (Optional) ---

# This cell is optional. It explores how changing the classification threshold
# (default 0.5) affects precision, recall, and F1 for the churn class.

if 'final_pipeline_model_rf_tuned' in locals() and 'test_df' in locals() and test_df.is_cached:
    print(f"\n--- Exploring Prediction Thresholds for Tuned RF on Target: {TARGET_COL} ---")

    # Get predictions if not already available (e.g., if best_predictions_rf was from a previous run)
    # It's better to re-run transform if unsure or if kernel restarted.
    # For now, assume best_predictions_rf (or final_predictions_rf_tuned) from Cell 8 is available.
    # If not, you'd do:
    # current_predictions_df = final_pipeline_model_rf_tuned.transform(test_df)

    if 'final_predictions_rf_tuned' not in locals():
        print("Predictions DataFrame ('final_predictions_rf_tuned') not found. Re-generating...")
        current_predictions_df = final_pipeline_model_rf_tuned.transform(test_df)
        current_predictions_df.persist() # Persist if re-generating for multiple threshold evals
    else:
        current_predictions_df = final_predictions_rf_tuned # Use existing if available

    print("Evaluating with different thresholds...")
    thresholds = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

    # Extract P(class=1) from probability vector
    # UDF to extract probability of positive class
    from pyspark.sql.functions import udf
    from pyspark.sql.types import FloatType
    first_element_udf = udf(lambda v: float(v[1]), FloatType()) # Assuming v[1] is P(class=1)

    if "probability" not in current_predictions_df.columns:
        print("ERROR: 'probability' column not found in predictions. Cannot adjust threshold.")
    else:
        predictions_with_p1 = current_predictions_df.withColumn("p1", first_element_udf(col("probability")))

        results_by_threshold = []
        for t in thresholds:
            # Apply new threshold
            predictions_thresholded = predictions_with_p1.withColumn(
                "prediction_adj",
                when(col("p1") >= t, 1.0).otherwise(0.0)
            )

            tp = predictions_thresholded.filter((col(TARGET_COL) == 1) & (col("prediction_adj") == 1.0)).count()
            fp = predictions_thresholded.filter((col(TARGET_COL) == 0) & (col("prediction_adj") == 1.0)).count()
            fn = predictions_thresholded.filter((col(TARGET_COL) == 1) & (col("prediction_adj") == 0.0)).count()
            tn = predictions_thresholded.filter((col(TARGET_COL) == 0) & (col("prediction_adj") == 0.0)).count()

            recall_c1 = tp / (tp + fn) if (tp + fn) > 0 else 0.0
            precision_c1 = tp / (tp + fp) if (tp + fp) > 0 else 0.0
            f1_c1 = 2 * (precision_c1 * recall_c1) / (precision_c1 + recall_c1) if (precision_c1 + recall_c1) > 0 else 0.0
            accuracy_overall = (tp + tn) / (tp + tn + fp + fn) if (tp + tn + fp + fn) > 0 else 0.0

            print(f"Threshold: {t:.2f} | Recall(C1): {recall_c1:.4f} | Precision(C1): {precision_c1:.4f} | F1(C1): {f1_c1:.4f} | Accura
            results_by_threshold.append({
                "Threshold": t, "Recall_Class1": recall_c1, "Precision_Class1": precision_c1,
                "F1_Class1": f1_c1, "Accuracy": accuracy_overall
            })

        # Plot Precision-Recall vs. Threshold
        if results_by_threshold:
            threshold_results_pd = pd.DataFrame(results_by_threshold)
            plt.figure(figsize=(10,6))
            plt.plot(threshold_results_pd["Threshold"], threshold_results_pd["Precision_Class1"], label="Precision (Class 1)", marker='
            plt.plot(threshold_results_pd["Threshold"], threshold_results_pd["Recall_Class1"], label="Recall (Class 1)", marker='x')
            plt.plot(threshold_results_pd["Threshold"], threshold_results_pd["F1_Class1"], label="F1-Score (Class 1)", marker='s')
            plt.xlabel("Prediction Threshold for Class 1")
            plt.ylabel("Score")
            plt.title(f"Precision, Recall, F1 for Churners vs. Threshold - {TARGET_COL}")
            plt.legend()
            plt.grid(True)
            plt.show()

    # Unpersist if we created current_predictions_df specifically here and persisted it
    if 'current_predictions_df' in locals() and current_predictions_df is not final_predictions_rf_tuned and current_predictions_df.is_
        current_predictions_df.unpersist()
        print("Unpersisted temporary predictions DataFrame for thresholding.")
else:
    print("Skipping Threshold Adjustment: Best tuned RF model or test_df not found / cached.")
```

```
--- Exploring Prediction Thresholds for Tuned RF on Target: Is_Churned_Engage_90Days ---
Evaluating with different thresholds...
Threshold: 0.10 | Recall(C1): 0.9745 | Precision(C1): 0.3167 | F1(C1): 0.4780 | Accuracy: 0.9653
Threshold: 0.20 | Recall(C1): 0.9203 | Precision(C1): 0.3819 | F1(C1): 0.5398 | Accuracy: 0.9744
Threshold: 0.30 | Recall(C1): 0.8111 | Precision(C1): 0.4706 | F1(C1): 0.5956 | Accuracy: 0.9820
Threshold: 0.40 | Recall(C1): 0.6326 | Precision(C1): 0.5665 | F1(C1): 0.5977 | Accuracy: 0.9861
Threshold: 0.50 | Recall(C1): 0.4507 | Precision(C1): 0.6363 | F1(C1): 0.5277 | Accuracy: 0.9868
Threshold: 0.60 | Recall(C1): 0.2827 | Precision(C1): 0.7131 | F1(C1): 0.4049 | Accuracy: 0.9864
Threshold: 0.70 | Recall(C1): 0.0775 | Precision(C1): 0.8261 | F1(C1): 0.1418 | Accuracy: 0.9847
Threshold: 0.80 | Recall(C1): 0.0103 | Precision(C1): 0.7921 | F1(C1): 0.0203 | Accuracy: 0.9838
Threshold: 0.90 | Recall(C1): 0.0000 | Precision(C1): 0.0000 | F1(C1): 0.0000 | Accuracy: 0.9837
```




```
# --- 13. Summarize Model Performance for the Current Target ---

# This cell assumes metrics from the BEST model (e.g., final tuned RF) are available as Python variables.
# E.g., roc_auc_final_rf, pr_auc_final_rf, accuracy_final_rf, f1_final_rf, recall_class1_final_rf, precision_class1_final_rf

if 'TARGET_COL' in globals() and \
   'roc_auc_final_rf' in locals() and 'pr_auc_final_rf' in locals() and \
   'accuracy_final_rf' in locals() and 'f1_final_rf' in locals() and \
   'recall_class1_final_rf' in locals() and 'precision_class1_final_rf' in locals():

    print(f"\n--- Performance Summary for Target: {TARGET_COL} (Best Model: Tuned Random Forest) ---")
    print(f"  AUC-ROC: {roc_auc_final_rf:.4f}")
    print(f"  AUC-PR: {pr_auc_final_rf:.4f}")
    print(f"  Overall Accuracy: {accuracy_final_rf:.4f}")
    print(f"  Weighted F1-Score: {f1_final_rf:.4f}")
    print("  ---------------------------------------------------")
    print(f"  Metrics for Churners (Class 1):")
    print(f"    Recall (Sensitivity): {recall_class1_final_rf:.4f}")
    print(f"    Precision: {precision_class1_final_rf:.4f}")
    calculated_f1_class1 = 0.0
    if (precision_class1_final_rf + recall_class1_final_rf) > 0:
        calculated_f1_class1 = 2 * (precision_class1_final_rf * recall_class1_final_rf) / (precision_class1_final_rf + recall_class1_fina
    print(f"    F1-Score (calculated): {calculated_f1_class1:.4f}")
    print("  ---------------------------------------------------")

    # You could also add confusion matrix components here if needed
    # tp_final_rf, fp_final_rf, fn_final_rf (should be in scope from Cell 8 evaluation part)
    if 'tp_final_rf' in locals() and 'fp_final_rf' in locals() and 'fn_final_rf' in locals():
        total_class1_actual = tp_final_rf + fn_final_rf
        total_class0_actual = test_df.count() - total_class1_actual # Approx.
        print(f"  Confusion Matrix Values (Class 1 is Churn):")
        print(f"    True Positives (Churned, Predicted Churned): {tp_final_rf}")
        print(f"    False Positives (Not Churned, Predicted Churned): {fp_final_rf}")
        print(f"    False Negatives (Churned, Predicted Not Churned): {fn_final_rf}")
        # TN = total_test - TP - FP - FN
        if 'test_count_loaded' in locals(): # from cell 5
            tn_final_rf = test_count_loaded - tp_final_rf - fp_final_rf - fn_final_rf
            print(f"    True Negatives (Not Churned, Predicted Not Churned): {tn_final_rf}")
else:
    print("Cannot generate summary: Required metric variables not found. Ensure Cell 8 (RF Tuning evaluation) ran successfully.")
```

```
# --- Ready to iterate for a new TARGET_COL by going back to Cell 3 ---
print("\n--- End of Modeling for Current Target ---")
print("To model another churn window, modify TARGET_COL in Cell 3 and re-run from Cell 3 onwards.")

# Optional: Unpersist train_df and test_df if truly done with this modeling session
# Or keep them if you plan to immediately switch TARGET_COL and re-run.
# if 'train_df' in locals() and train_df.is_cached: train_df.unpersist()
# if 'test_df' in locals() and test_df.is_cached: test_df.unpersist()
# print("Unpersisted train_df and test_df.")

# spark.stop() # Only stop Spark if completely done with the notebook.
```

```
--- Performance Summary for Target: Is_Churned_Engage_90Days (Best Model: Tuned Random Forest) ---
  AUC-ROC: 0.9914
  AUC-PR: 0.6077
  Overall Accuracy: 0.9868
  Weighted F1-Score: 0.9857
  ---------------------------------------------------
  Metrics for Churners (Class 1):
    Recall (Sensitivity): 0.4507
    Precision: 0.6363
    F1-Score (calculated): 0.5277
  ---------------------------------------------------
  Confusion Matrix Values (Class 1 is Churn):
    True Positives (Churned, Predicted Churned): 18587
    False Positives (Not Churned, Predicted Churned): 10623
    False Negatives (Churned, Predicted Not Churned): 22650
    True Negatives (Not Churned, Predicted Not Churned): 2473811

--- End of Modeling for Current Target ---
To model another churn window, modify TARGET_COL in Cell 3 and re-run from Cell 3 onwards.
```