NUS Datathon 2025

Category A

(Group 8)

Final Report

Team Members

Muhammad Abdul Rauf Bin Abdul Malik

Tan Teck Hwe Damaen

Tay Wei Ting

Introduction:

In today's digital age, financial institutions are increasingly leveraging data science to enhance customer experience and improve business performance. One key challenge in the financial advisory sector is ensuring that customers are matched with the right financial advisors who can best meet their unique needs. An optimal match can lead to better engagement, higher conversion rates, and greater customer satisfaction, ultimately driving revenue growth.

This datathon presents a unique opportunity for us to tackle this real-world challenge by developing a data-driven model that recommends the most suitable financial advisors for individual customers. By utilizing various machine learning techniques such as supervised learning, we aim to create an intelligent system that can effectively assign financial advisors to customers, optimizing engagement and improving the likelihood of successful policy conversions.

Financial advisors play a crucial role in guiding customers toward financial security by recommending suitable policies based on individual needs. However, due to the diversity in customer profiles and financial products, matching the right advisor to the right customer is a complex challenge. Current assignment processes often rely on manual selection or simplistic matching criteria, leading to inefficiencies, poor engagement, and missed opportunities.

By developing a robust data-driven model, we seek to optimize advisor-customer matching, leading to improved customer satisfaction, higher policy adoption rates, and enhanced revenue for financial institutions.

****

Dataset Overview:

We are given three datasets from Singlife - **Policy Information table, Client Information table and Agent information table.** Delving into the attributes of the three tables and understanding the columns:

****

1st Table: Policy Information Table

The Policy Information table contains data on individual insurance policies issued by agents. Key attributes include:

- **Identifiers:**
  - **chdrnum:** Unique identifier for each policy
  - **agntnum:** Identifier linking the policy to an agent
  - **secuityno:** Unique identifier for the customer associated with the policy

- **Policy Details:**
  - **occdate:** Date when the policy was issued
  - **annual_premium:** The annual premium paid for the policy
  - **product:** The specific product under the policy
  - **product_grp:** The product group classification

- **Policy Status Flags:**
  - **flg_main:** Indicates whether the policyholder is the primary holder
  - **flg_inforce:** Identifies policies that are currently active
  - **flg_cancel:** Identifies policies that have been canceled
  - **flg_expire:** Identifies policies that have expired
  - **flg_converted:** Identifies policies that have been converted

- **Customer Characteristics at Purchase:**
  - **cust_age_purchase_grp:** The age group of the customer at the time of policy purchase
  - **cust_tenure_at_purchase_grp:** The tenure category of the customer at purchase

****

2nd Table: Client Information Table

The Client Information table captures demographic and household information of clients. Key attributes include:

- **Customer Identifiers:**
  - **secuityno:** Unique identifier for the customer

- **Demographics:**
  - **cltsex:** Gender of the customer
  - **cltdob:** Date of birth of the customer
  - **marryd:** Marital status of the customer
  - **race_desc_map:** Race/ethnicity of the customer

- **Location and Household Information:**
  - **cltpcode:** Postal code of the customer's residence
  - **household_size:** Estimated household size based on postal code data
  - **economic_status:** Economic classification of the customer's residence area
  - **family_size:** Estimated family size based on postal code data
  - **household_size_grp:** Discretized household size category
  - **family_size_grp:** Discretized family size category

****

3rd Table: Agent Information Table

The Agent Information table provides insights into the agents who are responsible for selling and managing insurance policies. The key attributes include:

- **Agent Demographics:**
  - **agntnum:** Unique identifier for each agent
  - **agent_age:** Age of the agent
  - **agent_gender:** Gender of the agent
  - **agent_marital:** Marital status of the agent
  - **agent_tenure:** The number of years the agent has been with the company

- **Performance Metrics:**
  - **cnt_converted:** The count of policies converted by the agent
  - **annual_premium_cnvrt:** The total annual premium generated from converted policies

- **Policy Status Metrics:**
  - **pct_lapsed:** The percentage of policies that have lapsed
  - **pct_cancel:** The percentage of policies that have been canceled
  - **pct_inforce:** The percentage of policies that are currently active

- **Customer Segmentation Metrics:**
  - **pct_sx0_unknown, pct_sx1_male, pct_sx2_female:** The percentage of customers by gender that each agent handles
  - **pct_ag01_lt20, pct_ag02_20to24, ..., pct_ag10_60up:** The percentage of customers handled by the agent categorized into age groups

- **Product Specialization:**
  - **pct_prod_0_cnvrt, pct_prod_1_cnvrt, ..., pct_prod_9_cnvrt:** The percentage of each type of product sold by the agent
  - **agent_product_expertise:** A list of products that the agent is comfortable selling, based on feedback

- **Agent Segmentation:**
  - **cluster:** A predefined segment or cluster to which the agent belongs

****

Methodology:

Step 1: Importing Necessary Packages/Libraries

The first crucial step involves setting up the necessary environment and importing essential libraries. These libraries provide functions that facilitate data handling, preprocessing, visualization, and model building. The libraries that we will be importing include **pandas** for data cleaning and manipulation, **numpy** for numerical computation processes, **matplotlib** for data visualisation, **seaborn** for statistical data visualisation and **sklearn** for Machine Learning.

Importing the necessary libraries/packages for the data analysis process:

```
# Importing the necessary packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
```

****

Step 2: Reading The Parquet Files

Before performing any analysis, it is crucial to load the datasets into a structured format that can be manipulated efficiently. In this project, we have three essential datasets:

1. Policy Information – Contains details about different insurance policies.
2. Client Information – Holds records of clients, including demographic and personal details.

3. Agent Information – Provides data about agents responsible for handling policies and clients.

Since these datasets are stored in Parquet format, we will utilize the `pd.read_parquet()` method from the Pandas library to read them into DataFrames.

Example: Reading the Policy Information Parquet file

```
[50] # Extracting the policy information parquet file from Google Colab
     policy_info = '/content/drive/MyDrive/nus_policy_info_df.parquet'

     # Reading the policy information parquet file
     df_policy = pd.read_parquet(policy_info)

     # Viewing the first few rows of the policy information dataset
     df_policy.head()
```

After the code above is executed, the dataset is returned as the output, suggesting that the reading of the parquet files are successful. Similar approaches can be used for the other two dataframes - Client Information and Agent information.

Output of the Policy Information Table after file reading:

	chdrnum	agntnum	secuityno	occdate	annual_premium	product	flg_main	flg_rider	flg_inforce	flg_lapsed	flg_cancel	flg_expire	flg_converted	product_grp
0	PID:281	AIN:62	CIN:6957	2018-11-12	0.0	prod_8	1	0	1	0	0	0	1	PG:0
1	PID:280	AIN:63	CIN:2161	2024-02-22	7.0	prod_8	1	0	1	0	0	0	1	PG:0
2	PID:2577	AIN:63	CIN:16605	2024-08-28	423.0	prod_6	1	0	1	0	0	0	1	PG:0
3	PID:2578	AIN:63	CIN:16605	2024-08-27	217.0	prod_6	1	0	1	0	0	0	1	PG:0
4	PID:305	AIN:63	CIN:7917	2024-08-28	432.0	prod_6	1	0	1	0	0	0	1	PG:0

****

Step 3: Inspecting The Datasets

Once the datasets are loaded, the next critical step in the data analysis workflow is inspecting them. This involves understanding their structure, dimensions, and data types to ensure the data is ready for further analysis. The inspection process helps identify potential data quality issues such as missing values, inconsistent data types, and redundant information.

Inspection Methods Used:

1. **Examining Column Names**

   The `.columns` attribute of a DataFrame is used to extract the column names of the dataset. Knowing the column names is crucial for:

   - Understanding the structure and information contained within the dataset
   - Identifying key features for analysis and modeling
   - Detecting any unexpected or incorrectly named columns

2. **Checking Dataset Dimensions**

   The `.shape` attribute is used to determine the number of rows and columns in the dataset. This helps:

   - Understand the size of the dataset
   - Identify if the dataset is too large to process directly in memory
   - Confirm whether the dataset matches expected dimensions

3. **Examining Data Types and Non-Null Counts**

   The `.info()` method provides a summary of the dataset, including:

   - Data types of each column (e.g., `int64`, `float64`, `object`)
   - Non-null counts for each column, helping identify missing data
   - Memory usage, which can be optimized if necessary

Example: Inspecting the Policy Information Dataset

```
print("Policy Information Dataset: ")
print("\n")

# Examine the column names of the policy information dataset
print(f"Column Names: {list(df_policy.columns)}")
print("\n")

# Check the dimensions of the policy information dataset
print(f"Dimensions: {df_policy.shape}")
print("\n")

# Examine the data types of the policy information dataset
df_policy.info()
```

Output of the Policy Information Dataset after inspection:

```
Policy Information Dataset:


Column Names: ['chdrnum', 'agntnum', 'secuityno', 'occdate', 'annual_premium', 'product', 'flg_main', 'flg_rider', 'flg_inforce', 'flg_lapsed', 'flg_cancel', 'fl


Dimensions: (29503, 16)


<class 'pandas.core.frame.DataFrame'>
RangeIndex: 29503 entries, 0 to 29502
Data columns (total 16 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   chdrnum                      29503 non-null  object
 1   agntnum                      29503 non-null  object
 2   secuityno                    29503 non-null  object
 3   occdate                      29503 non-null  datetime64[us]
 4   annual_premium               29503 non-null  float64
 5   product                      29503 non-null  object
 6   flg_main                     29503 non-null  int64
 7   flg_rider                    29503 non-null  int64
 8   flg_inforce                  29503 non-null  int64
 9   flg_lapsed                   29503 non-null  int64
 10  flg_cancel                   29503 non-null  int64
 11  flg_expire                   29503 non-null  int64
 12  flg_converted                29503 non-null  int64
 13  product_grp                  29503 non-null  object
 14  cust_age_at_purchase_grp     29503 non-null  object
 15  cust_tenure_at_purchase_grp  29503 non-null  object
dtypes: datetime64[us](1), float64(1), int64(7), object(7)
memory usage: 3.6+ MB
```

****

Step 4: Observing The Summary Statistics Of Datasets

Analyzing the summary statistics of numerical columns is a crucial step in understanding the overall characteristics of a dataset. It provides insights into the central tendency, spread, and distribution of numerical data. In this section, we focus on calculating the summary statistics of the three datasets using the `.describe()` method in Pandas.

Example: Finding the summary statistics of Policy Information Dataset

```
print("Policy Information Dataset: ")
print("\n")

# Find the summary statistics of the policy information dataset
df_policy.describe()
```

Summary Statistics Output of Policy Information Dataset:

Policy Information Dataset:

	occdate	annual_premium	flg_main	flg_rider	flg_inforce	flg_lapsed	flg_cancel	flg_expire	flg_converted
count	29503	29503.000000	29503.0	29503.0	29503.000000	29503.000000	29503.0	29503.000000	29503.0
mean	2020-09-04 17:00:41.121242	1081.708652	1.0	0.0	0.759719	0.222249	0.0	0.018032	1.0
min	1993-10-01 00:00:00	0.000000	1.0	0.0	0.000000	0.000000	0.0	0.000000	1.0
25%	2017-05-25 00:00:00	0.000000	1.0	0.0	1.000000	0.000000	0.0	0.000000	1.0
50%	2022-07-21 00:00:00	127.000000	1.0	0.0	1.000000	0.000000	0.0	0.000000	1.0
75%	2024-04-09 00:00:00	528.000000	1.0	0.0	1.000000	0.000000	0.0	0.000000	1.0
max	2024-11-28 00:00:00	300039.600000	1.0	0.0	1.000000	1.000000	0.0	1.000000	1.0
std	NaN	4422.632426	0.0	0.0	0.427261	0.415764	0.0	0.133070	0.0

****

Step 5: Data Cleaning Of The Datasets

From the inspection of the three datasets, they are not in clean formats and there are several processes that we need to carry out to tidy the datasets to more readable formats.

Step 5.1: Standardizing Column Names

Consistency in column naming is a vital aspect of data preparation. Inconsistent column names, such as varying cases (uppercase, lowercase, or mixed case), can lead to challenges when performing data manipulations, merging datasets, or building machine learning pipelines.

Key Reasons for Standardizing Column Names:

1. **Data Consistency:** Uniform formatting of column names eliminates ambiguity and ensures consistency across different datasets
2. **Error Prevention:** Prevents case-sensitive errors during operations like dataset joins, merges, and comparisons
3. **Code Simplification:** Reduces the need to account for case differences in coding, making scripts cleaner and easier to maintain

Issue in the Datasets:

From the provided datasets, several column names, such as:

- `pct_SX0_unknown`
- `pct_AG01_lt20`
- `pct_AG10_60up`

contain mixed capitalization, which can lead to inconsistencies when interacting with other datasets or performing transformations.

The **pandas** library provides an efficient way to rename column names using the **.str.lower()** method. This method converts all column names in a DataFrame to lowercase.

Standardizing Column Names All To Lowercase:

```
# Standardize column names to lowercase to prevent mismatches
df_policy.columns = df_policy.columns.str.lower()
df_client.columns = df_client.columns.str.lower()
df_agent.columns = df_agent.columns.str.lower()
```

****

Step 5.2: Tidying Values To More Readable Formats

Readable data is essential for effective analysis, communication, and decision-making. This step focuses on tidying up column values in the three datasets by replacing or modifying values to make them more interpretable. Specifically, this involves removing extraneous text, symbols, or prefixes while retaining meaningful numerical or categorical information.

Issue in the Datasets:

Unnecessary Characters: Many values include prefixes, punctuation, or words that do not contribute to the analysis. For example:

- **AIN:9513** in the **agntnum** column could simply be **9513**
- **prod_8** in the **product** column could be replaced with **8**

Variations in formatting across values can lead to complications during aggregation, grouping, or filtering operations. Hence, it is best practice to remove unnecessary and irrelevant characters that do not contribute to the Machine Learning model's accuracy.

	chdrnum	agntnum	secuityno
0	PID:281	AIN:62	CIN:6957
1	PID:280	AIN:63	CIN:2161
2	PID:2577	AIN:63	CIN:16605
3	PID:2578	AIN:63	CIN:16605
4	PID:305	AIN:63	CIN:7917

	chdrnum	agntnum	secuityno
0	281	62	6957
1	280	63	2161
2	2577	63	16605
3	2578	63	16605
4	305	63	7917

(Original Uncleaned Values) —----------> (New Cleaned Values)

To ensure that the column values are more readable, we need to **remove the necessary punctuation and words, but leave the numbers**.To clean the column values, **regular expressions (regex)** are employed. Regex is a powerful tool for pattern matching and replacement, which allows for efficient and targeted modifications.

Regex Expression When Cleaning Values In Columns:

```
# Regex expression function

# Tidying the policy dataset
policy_clean = ['chdrnum', 'agntnum', 'secuityno', 'product', 'product_grp']

for col in policy_clean:
    df_policy[col] = df_policy[col].str.replace(r'[^0-9]', '', regex = True)

# Tidying the client dataset
df_client['secuityno'] = df_client['secuityno'].str.replace(r'[^0-9]', '', regex = True)

# Tidying the agent dataset
df_agent['agntnum'] = df_agent['agntnum'].str.replace(r'[^0-9]', '', regex = True)
```

****

Step 5.3: Tidying On More Complex Columns

While regular expressions (regex) are powerful tools for data cleaning, some columns may require more nuanced approaches due to their complex nature. Columns such as **`cust_age_at_purchase_grp`**, **`cust_tenure_at_purchase_grp`**, **`household_size_grp`**, and **`family_size_grp`** contain values that encode specific ranges or groups, making direct regex transformations insufficient.

Characteristics of Complex Columns:

1. **Categorical Ranges:**
    - Values like `20-24` (representing age ranges) in `cust_age_at_purchase_grp` or `1-4 years` in `cust_tenure_at_purchase_grp` require careful interpretation
2. **Non-Uniform Representations:**
    - Values may include inconsistent use of symbols (e.g., `lt20`, `20to24`, `60up` for the `cust_age_at_purchase_grp` column).

Although regex is effective for pattern matching, it struggles to handle complex semantic interpretations as transformations require knowledge of context or relationships between values that regex is unable to provide.

Instead, we should first examine the various categories of these complex columns and what each value represents. We can use the `.unique()` method to examine all the possible values of each column.

Obtaining The Unique Column Values:

```
# Identify unique values in the columns
unique_age_groups = df_policy['cust_age_at_purchase_grp'].unique()
unique_tenure_groups = df_policy['cust_tenure_at_purchase_grp'].unique()
unique_household_sizes = df_policy['household_size_grp'].unique()
unique_family_sizes = df_policy['family_size_grp'].unique()

print("Unique Age Groups:", unique_age_groups)
print("Unique Tenure Groups:", unique_tenure_groups)
print("Unique Household Sizes:", unique_household_sizes)
print("Unique Family Sizes:", unique_family_sizes)
```

After obtaining the list of column values, we can find out what each value represents and at the same time, know the new encoded value when the column is encoded. We can determine the new encoded numbers by **mapping the created encoded values to the original unmodified column values,** which can be achieved using a mapping dictionary.

Example: Information and Encoded Column Values Of A Complex Column

cust_age_at_purchase_grp information:

Original Column Value	Revised Column Value	Information
AG00_missing	0	Missing Age Information
AG01_lt20	1	Below 20 years old
AG02_20to24	2	From 20 to 24 years old
AG03_25to29	3	From 25 to 29 years old
AG04_30to34	4	From 30 to 34 years old
AG05_35to39	5	From 35 to 39 years old
AG06_40to44	6	From 40 to 44 years old
AG07_45to49	7	From 45 to 49 years old
AG08_50to54	8	From 50 to 54 years old
AG09_55to59	9	From 55 to 59 years old
AG10_60up	10	Over 60 years old

Example: Mapping The New Encoded Value To The Original Column Values

```
# Create a mapping dictionary for cust_age_at_purchase_grp
mapping_dict_one = {"AG00_missing": 0, "AG01_lt20": 1, "AG02_20to24": 2,
                    "AG03_25to29": 3, "AG04_30to34": 4, "AG05_35to39": 5,
                    "AG06_40to44": 6, "AG07_45to49": 7, "AG08_50to54": 8,
                    "AG09_55to59": 9, "AG10_60up": 10}

# Apply mapping to cust_age_at_purchase_grp
df_policy["cust_age_at_purchase_grp"] = df_policy["cust_age_at_purchase_grp"].map(mapping_dict_one)
```

****

Step 5.4: Checking and removing all NA values

Missing or NA (Not Available) values are common in datasets and can hinder data analysis and machine learning processes. Identifying and appropriately handling these values is a crucial part of data preprocessing. To understand the extent and distribution of missing values in the datasets, we use the **`.isna().sum()`** method, which provides a column-wise count of NA values.

Finding The Presence Of NA's For The Three Datasets:

```
# Checking for missing values in each dataset
print("Missing values in Policy Information Dataset:")
print(df_policy.isna().sum())

print("\nMissing values in Client Information Dataset:")
print(df_client.isna().sum())

print("\nMissing values in Agent Information Dataset:")
print(df_agent.isna().sum())
```

We observe that there are no NA values for all columns of the Policy Information Dataset. However, for the other two datasets - Client Information Dataset and Agent Information Dataset, there are NA values. For instance, several columns of the Client Information Dataset, including **`cltdob`**, **`race_desc_map`**, **`cltpcode`**, **`household_size`**, **`economic_status`** and **`family_size`** have NA values (**`.isna().sum()`** does not evaluate to 0)

Example: Result Of Number Of NA's For Client Information Dataset

```
secuityno              0
cltsex                 0
cltdob                 6
marryd                 0
race_desc_map         10
cltpcode             156
household_size       343
economic_status      343
family_size          343
household_size_grp     0
family_size_grp        0
dtype: int64
```

There are several strategies that we can adopt to handle the NA values, such as **using imputation** or **removing records containing the NA values totally.**

Why Not Use Imputation?

In this context, imputing missing values using statistical methods such as mean, median, or mode is not recommended because:

1. **Accuracy Concerns:** Imputed values may not accurately reflect the true values, introducing biases into the data.
2. **Impact on Machine Learning Models:** Imputed values can distort patterns in the data, negatively affecting the model's performance.

Hence, to maintain the integrity of the data, rows containing missing values should be removed instead. To remove the records containing NA values, we can use the `.dropna()` method.

Removing Rows With NA For Client And Agent Datasets:

```
# Remove rows with NA values for the df_client dataset
df_client.dropna(inplace = True)

# Remove rows with NA values for the df_agent dataset
df_agent.dropna(inplace = True)
```

****

Step 5.5: Removing All Possible Duplicates In The Datasets

Next, we should remove all the duplicate records in the three datasets (if any). Duplicate records in a dataset can result in redundancy, inefficiency, and skewed results in data analysis and machine learning processes. It is critical to identify and remove such duplicate records to maintain data quality.

To identify duplicate rows, the `.duplicated().sum()` method is used. This method calculates the total number of duplicate rows in the dataset. A value of 0 indicates no duplicate rows are present while a non-zero value indicates the presence of duplicates, requiring further action to remove them.

Special Case for Agent Dataset - Handling the `agent_product_expertise` Column:

In the agent dataset, the `agent_product_expertise` column contains lists, making it incompatible with the `.duplicated()` method. Attempting to check for duplicates with this column intact will result in a `TypeError`. To resolve this, the column is temporarily dropped before checking for duplicates.

Checking For Duplicates For The Three Datasets:

```
# Check for duplicate rows in each dataset
policy_duplicates = df_policy.duplicated().sum()
client_duplicates = df_client.duplicated().sum()

# Remove the agent_product_expertise column for the time being (while checking
# for duplication)
new_df = df_agent.drop('agent_product_expertise', axis = 1)
agent_duplicates = new_df.duplicated().sum()

# Print the results
print(f"Policy Dataset: {policy_duplicates} duplicate rows found")
print(f"Client Dataset: {client_duplicates} duplicate rows found")
print(f"Agent Dataset: {agent_duplicates} duplicate rows found")
```

Output To Check For Any Duplicated Rows:

```
Policy Dataset: 0 duplicate rows found
Client Dataset: 0 duplicate rows found
Agent Dataset: 0 duplicate rows found
```

From the results, **0 duplicate rows** are found in all three datasets, suggesting that the **records are all unique**. By identifying and removing duplicates, we ensure the integrity and uniqueness of the datasets. Special considerations, such as handling columns with incompatible data types (eg. lists), are necessary to avoid errors. With duplicates eliminated, the datasets are now streamlined and ready for further analysis and modeling.

****

Step 5.6: Converting Columns To The Right Data Type

Next, we need to convert the data type of several columns of the respective datasets to the correct type. This is especially so after the categorical labelling and removal of letters and symbols in Steps 5.2 and 5.3. Data type mismatches can hinder the analysis process and negatively impact the performance of machine learning models. It is essential to ensure that columns have the correct data types to allow for efficient computation and accurate data handling.

For instance, there are several columns in the Policy Information Dataset which require a conversion of data type from **object** to **int32**. The same approach is performed for the other two datasets.

Example: Policy Information Dataset

Column	Current Type	Desired Type
chdrnum	object	int32
agntnum	object	int32
secuityno	object	int32
product	object	int32
product_grp	object	int32

cust_age_at_purchase_grp	object	int32
cust_tenure_at_purchase_grp	object	int32

We can convert the columns with incorrect data types to the appropriate ones using Python's **`.astype()`** method. For ease of implementation, the columns requiring data type conversions are stored in separate lists for each dataset.

Converting Columns To The Right Data Type:

```
# Define columns to convert for each dataset
policy_columns = ["chdrnum", "agntnum", "secuityno", "product", "product_grp",
                  "cust_age_at_purchase_grp", "cust_tenure_at_purchase_grp"]

client_columns = ["secuityno", "household_size", "economic_status", "family_size",
                  "household_size_grp", "family_size_grp"]

agent_columns = ["agntnum"]

# Convert columns to int32 if they exist in the DataFrame
for col in policy_columns:
    if col in df_policy.columns:
        df_policy[col] = df_policy[col].astype("int32")

for col in client_columns:
    if col in df_client.columns:
        df_client[col] = df_client[col].astype("int32")

for col in agent_columns:
    if col in df_agent.columns:
        df_agent[col] = df_agent[col].astype("int32")
```

****

Step 5.7: Removing Unnecessary Columns

Feature selection and dimensionality reduction are critical steps in preparing data for machine learning. Removing unnecessary or irrelevant columns helps streamline the analysis process, improves computational efficiency, and reduces the risk of overfitting in machine learning models.

For some of the datasets, there are a couple of features that should be removed and not be included in the Machine Learning models due to several reasons.

Identifying Unnecessary Columns

1. **Client Dataset**

- **Column: `cltpcode`**
  - **Reason for Removal:** The postal code of the client is unrelated to the target variable, which is the agent's conversion success rate. Retaining this column adds noise and may adversely affect the model's performance.

2. **Policy Dataset**
   - **Column: `flg_converted`**
     - **Reason for Removal:** The descriptive statistics reveal that all rows in this column have a value of **1**. Since this feature does not vary, it is non-informative and cannot help differentiate or classify outcomes in the machine learning model. Keeping this column adds no value to the analysis or prediction tasks.

Other repetitive columns include **`family_size`** and **`household_size`** columns in the client dataset which are repeated in the **`family_size_grp`** and **`household_size_grp`** columns respectively.

The **`pandas .drop()`** function is used to remove unnecessary columns from the datasets. This function allows for the removal of specific columns by their names.

Dropping The Unnecessary Columns:

```
# Drop the cltpcode column from the client dataset
df_client.drop(columns = ['cltpcode', 'family_size', 'household_size'], inplace = True)

# Drop the flg_converted  column from the policy dataset
df_policy.drop(columns = ['flg_converted'], inplace = True)
```

Removing unnecessary columns such as **`cltpcode`** and **`flg_converted`** optimizes the datasets for further analysis and machine learning tasks. This step eliminates irrelevant and redundant information, ensuring that the datasets are more focused and efficient. These changes will contribute to building a more accurate and interpretable machine learning model.

****

Step 6: Data Transformation Of The Datasets

After cleaning the datasets, several transformations are required before the final dataset is deemed suitable to be used for the Machine Learning model.

Step 6.1: Label Encoding For Categorical Columns

In machine learning, categorical variables must be converted into numerical formats that models can process. Label encoding is one approach to achieve this, assigning a unique integer to each category. This step describes the identification, analysis, and encoding of categorical columns in the Client and Agent Information datasets.

Datasets and Relevant Columns:

1. **Policy Information Dataset:**
    - No categorical columns requiring encoding

2. **Client Information Dataset:**
    - `cltsex`**:** Represents the client's gender
    - `marryd`**:** Indicates the marital status of the client
    - `race_desc_map`**:** Provides information about the race/ethnicity of the client

3. **Agent Information Dataset:**
    - `agent_gender`**:** Represents the agent's gender
    - `agent_marital`**:** Indicates the marital status of the agent

However, before immediately performing the encoding, we need to ask ourselves two questions - **what are the values in these columns that require encoding** and **how should we encode these values?**

Similar to Step 5.3, to obtain the various values of the encoded columns, we need to use the `.unique()` method on the respective columns.

Exploring The Value Types Of The Columns That Require Encoding:

```
# Exploring the unique values of the encoded columns of the three datasets
print("Unencoded columns unique values: ")
print("\n")

# 1st column: cltsex column
print(f"Unique values of cltsex column: {df_client['cltsex'].unique()}")

# 2nd column: marryd column
print(f"Unique values of marryd column: {df_client['marryd'].unique()}")

# 3rd column: race_desc_map column
print(f"Unique values of race_desc_map column: {df_client['race_desc_map'].unique()}")

# 4th column: agent_gender column
print(f"Unique values of agent_gender column: {df_agent['agent_gender'].unique()}")

# 5th column: agent_marital column
print(f"Unique values of agent_marital column: {df_agent['agent_marital'].unique()}")
```

The Lists Of Possible Values For The Categorical Columns:

```
Unencoded columns unique values:


Unique values of cltsex column: ['F' 'M']
Unique values of marryd column: ['M' 'S' 'D' 'U' 'W' 'P']
Unique values of race_desc_map column: ['Chinese' 'Others' 'Indian' 'Malay']
Unique values of agent_gender column: ['M' 'F' 'U']
Unique values of agent_marital column: ['M' 'D' 'S' 'U' 'W']
```

To gain a comprehensive understanding of the possible value types and their meanings, we need to analyze each category systematically. For example, the `marryd` column, which represents the marital status of the client, contains six distinct values: **'M', 'S', 'D', 'U', 'W', and 'P'**.

To ensure clarity and consistency, we can construct a table that maps each original column value to its corresponding meaning and its newly assigned encoded value. The encoding will be systematically ordered based on the alphabetical arrangement of the original unencoded values. Similar tables can be constructed for the other four categorical columns - `cltsex`, `race_desc_map`, `agent_gender` and `agent_marital`.

Example: Table To Illustrate The Values Of The `marryd` Column

Unencoded Column Value	What It Represents	Encoded Column Value
D	Divorced	0

M	Married	1
P	Partnered	2
S	Single	3
U	Unknown	4
W	Widowed	5

For categorical columns such as **`cltsex`** (Male, Female), **`marryd`** (Single, Widowed, Divorced, etc.), and **`race_desc_map`** (Chinese, Malay, Indian, White, etc.), there is no intrinsic or meaningful ordinal relationship between the categories. Since these categorical values represent distinct groups without a natural ranking, we should apply **Label Encoding** using **`label_encoder.fit_transform()`**.

This method assigns a unique numerical value to each category, converting categorical data into a format suitable for machine learning models. While Label Encoding is typically used for ordinal data, in this case, it provides a simple numerical representation of categorical variables.

Initializing The Label Encoder From Scikit-Learn:

```
# Import the Label Encoder from scikit-learn
from sklearn.preprocessing import LabelEncoder

# Initialize the LabelEncoder
label_encoder = LabelEncoder()
```

Perform Label Encoding On The Categorical Variables:

```
# Label Encode the relevant columns

# Label Encode cltsex column
df_client['cltsex'] = label_encoder.fit_transform(df_client['cltsex'])

# Label Encode marryd column
df_client['marryd'] = label_encoder.fit_transform(df_client['marryd'])

# Label Encode race_desc_map column
df_client['race_desc_map'] = label_encoder.fit_transform(df_client['race_desc_map'])

# Label Encode agent_gender column
df_agent['agent_gender'] = label_encoder.fit_transform(df_agent['agent_gender'])

# Label Encode agent_marital column
df_agent['agent_marital'] = label_encoder.fit_transform(df_agent['agent_marital'])
```

****

Step 6.2: Applying Log Transformation For Highly Skewed Numerical Variables

From the descriptive statistics of the policy and agent datasets, the columns **`annual_premium`** and **`annual_premium_cnvrt`**, respectively, exhibited highly skewed distributions with extreme values. Such distributions can adversely affect the performance of machine learning models by introducing bias and instability.

For instance, the **`annual_premium`** column of the Policy Information Dataset has a maximum value of over 300000 while more than 25% of the data points have values of 0 (as shown in the screenshot below, where 25% is still 0.0). This shows that the values of **`annual_premium`** are highly skewed with a wide range of values, many of which are at the low end of the range (eg. between 0 and 1000 - includes more than 75% of the data points)

Descriptive Statistics Of The **`annual_premium`** column:

	occdate	annual_premium
count	29503	29503.000000
mean	2020-09-04 17:00:41.121242	1081.708652
min	1993-10-01 00:00:00	0.000000
25%	2017-05-25 00:00:00	0.000000
50%	2022-07-21 00:00:00	127.000000
75%	2024-04-09 00:00:00	528.000000
max	2024-11-28 00:00:00	300039.600000
std	NaN	4422.632426

To address this, a log transformation using the `np.log1p()` function is applied. This transformation is defined as `log1p(x) = log(1 + x)` and is suitable for variables that may contain zero values, as it avoids undefined logarithmic calculations for zero. This method normalizes the data, reduces skewness, and stabilizes variance, making the datasets more suitable for downstream machine learning tasks.

Performing Log Transformations On The Skewed Numerical Columns:

```
# Log transformation of annual_premium in df_policy
df_policy['annual_premium'] = np.log1p(df_policy['annual_premium'])

# Log transformation of annual_premium_cnvrt in df_agent
df_agent['annual_premium_cnvrt'] = np.log1p(df_agent['annual_premium_cnvrt'])
```

After applying the log transformation, the descriptive statistics of the columns were reviewed to assess the effectiveness of the normalization process.

Results After Log Transformations:

1. **Policy Dataset (`annual_premium`):**
    - Before Transformation:
        - **Min: 0**
        - **Max: >300,000**
        - **Median: Significantly skewed toward lower values**
    - After Transformation:
        - **Min: 0**
        - **Max: ~12**
        - **Median: Stabilized, with reduced skewness**

2. **Agent Dataset (`annual_premium_cnvrt`):**
    - Before Transformation:
        - **Min: 0**
        - **Max: > 3e7**
        - **Median: Strongly skewed to the left**
    - After Transformation:
        - **Min: 0**
        - **Max: ~17**
        - **Median: Normalized to a more uniform distribution**

****

Step 6.3: Handling Lists Of Values From `agent_product_expertise` column

The `agent_product_expertise` column in the agent dataset contains lists of product codes for which each agent has expertise. While this format provides the necessary information, it is not suitable for analysis or machine learning tasks. **Converting this data from a long format (lists) to a wide format (binary encoding)** makes it more readable and useful for downstream tasks such as feature engineering and predictive modeling.

Objective

Transform the `agent_product_expertise` column into individual binary columns, one for each product, where:

- `1` indicates the agent's expertise in the product.
- `0` indicates no expertise in the product.

The resulting columns will be named `prod_1_expert`, `prod_2_expert`, `...`, `prod_n_expert`, where `n` is the total number of unique products. This transformation can be done using the `MultiLabelBinarizer` class from the `sklearn.preprocessing` package.

Step 1: Initialize the MultiLabelBinarizer

The `MultiLabelBinarizer` class from `sklearn.preprocessing` is ideal for handling columns with list-like values. This tool converts a column of lists into multiple binary columns.

Step 2: Create New Columns

The `fit_transform` method of the `MultiLabelBinarizer` is applied to the `agent_product_expertise` column. This generates a matrix where each row corresponds to a record, and each column indicates whether the agent is an expert in a particular product.

Step 3: Concatenate the New Columns with the Original Dataset

The new dataframe containing the binary columns is concatenated with the original `df_agent` dataframe.

Step 4: Remove the Original `agent_product_expertise` Column

After creating the binary columns, the original **`agent_product_expertise`** column is no longer needed and should be dropped.

Step 5: Handle Missing Values

During the transformation process, any rows containing missing values (**`NA`**) in the newly created binary columns are removed to maintain dataset consistency.

Code Containing The 5 Steps Of The `MultiLabelBinarizer` Process:

```
from sklearn.preprocessing import MultiLabelBinarizer

# Initialize MultiLabelBinarizer
mlb = MultiLabelBinarizer()
expertise_encoded = mlb.fit_transform(df_agent['agent_product_expertise'])

# Create a new DataFrame with feature names
expertise_df = pd.DataFrame(expertise_encoded,
                            columns=[f"{prod}_expert" for prod in mlb.classes_])

# Concatenate with original DataFrame
df_agent = pd.concat([df_agent, expertise_df], axis=1)

# Drop original column if needed
df_agent.drop(columns=['agent_product_expertise'], inplace=True)

# Remove all the NA values subsequently
df_agent = df_agent.dropna()
```

****

Step 6.4: Joining The Three Datasets Into One

Before implementing a machine learning model, it is essential to consolidate the data from multiple sources into a single structured dataset. The three datasets—**Policy Information, Client Information, and Agent Information**—contain complementary details about policies, clients, and agents. To build a robust ML model, these datasets must be joined efficiently while ensuring data integrity.

We will merge the datasets using **inner joins** to ensure that only records present in all datasets are retained, preventing any missing values (**`NA`**). The merging process follows a two-step approach:

1. **First Join**: Merge the **Policy Information Dataset** with the **Client Information Dataset** using the common column **`secuityno`**.
2. **Second Join**: Merge the **resulting dataset** with the **Agent Information Dataset** using the common column **`agntnum`**.

We use **pandas' `pd.merge()` method** with an **inner join** to ensure that only records with complete information across all three datasets are included.

Two-Step Merging Process To Join Three Datasets Into One:

```
# Step 1: Merge the Policy Information Dataset and Client Information Dataset
partial_merged_df = pd.merge(df_policy, df_client, on= 'secuityno', how = 'inner')

# Step 2: Merge the resulting dataset and Agent Information Dataset
final_df = pd.merge(partial_merged_df, df_agent, on = 'agntnum', how = 'inner')

# Viewing the final resultant merged dataframe
final_df.head()
```

After executing the necessary joins, our final dataset consists of **62 columns**, all of which have been properly encoded. This transformation enhances readability and ensures that the data is in a structured format suitable for further analysis and machine learning applications.

Final Cleaned Transformed Dataset For Analysis:

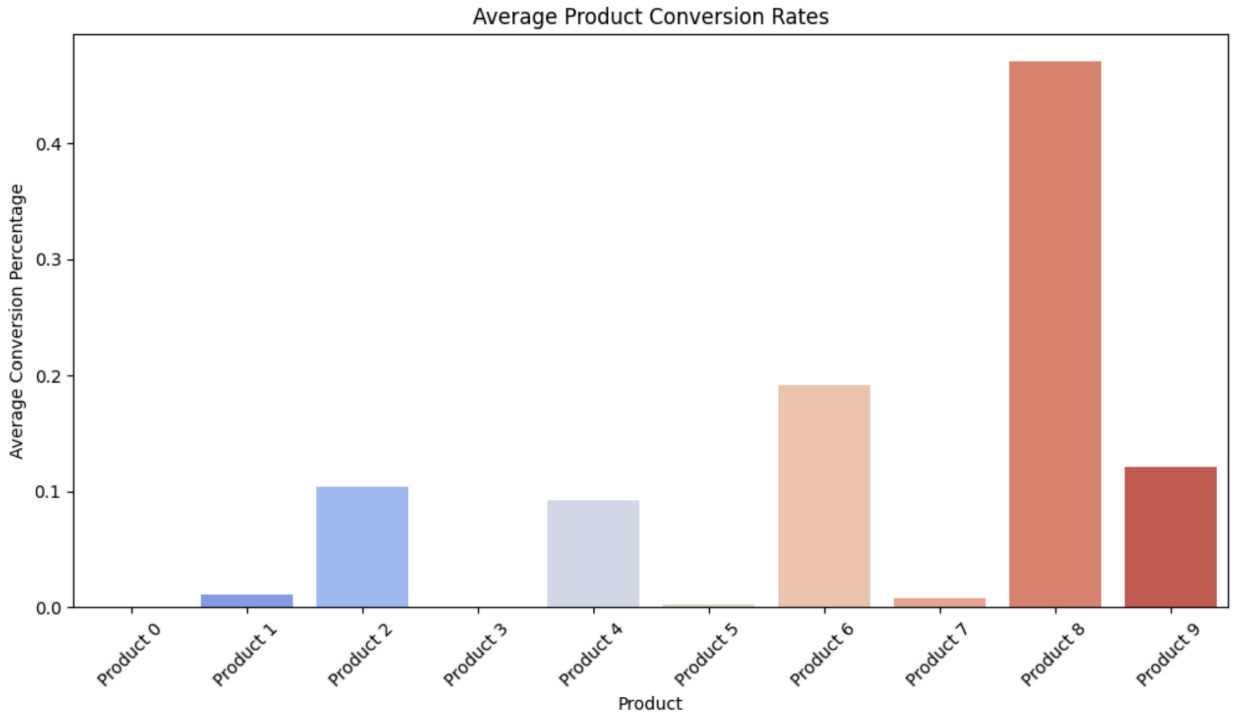
	chdrnum	agntnum	secuityno	occdate	annual_premium	product	flg_main	flg_rider	flg_inforce	flg_lapsed	...	pct_ag09_55to59	pct_ag10_60up	cluster	prod_
0	281	62	6957	2018-11-12	0.000000	8	1	0	1	0	...	0.000000	0.000000	1.0	
1	280	63	2161	2024-02-22	2.079442	8	1	0	1	0	...	0.019956	0.011842	1.0	
2	2577	63	16605	2024-08-28	6.049733	6	1	0	1	0	...	0.019956	0.011842	1.0	
3	2578	63	16605	2024-08-27	5.384495	6	1	0	1	0	...	0.019956	0.011842	1.0	
4	305	63	7917	2024-08-28	6.070738	6	1	0	1	0	...	0.019956	0.011842	1.0	

5 rows × 62 columns

****

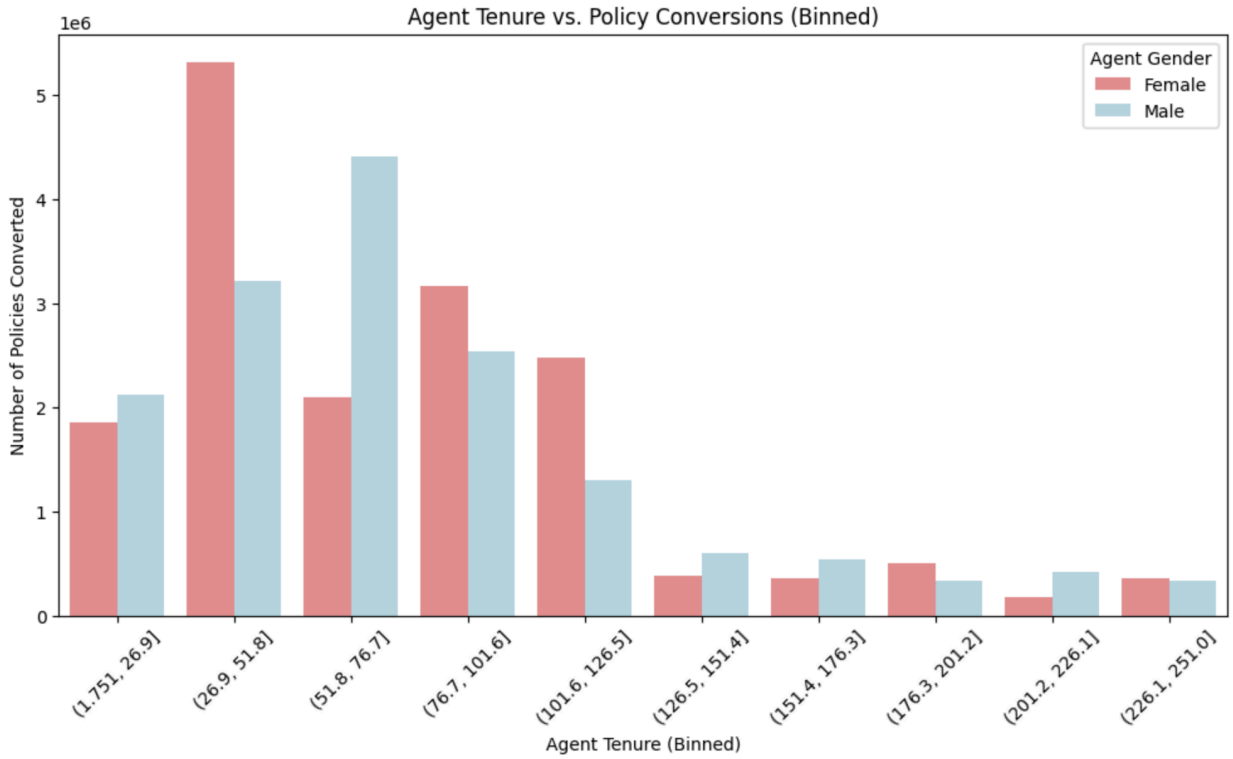
Step 7: Visualising the Cleaned Data

After cleaning and transforming the dataset, our next objective was to visualize key aspects of the data to gain a better understanding of the underlying patterns. We focused on analyzing the interrelationships between policy conversions, products, and agents. Below are the key insights derived from the visualizations.



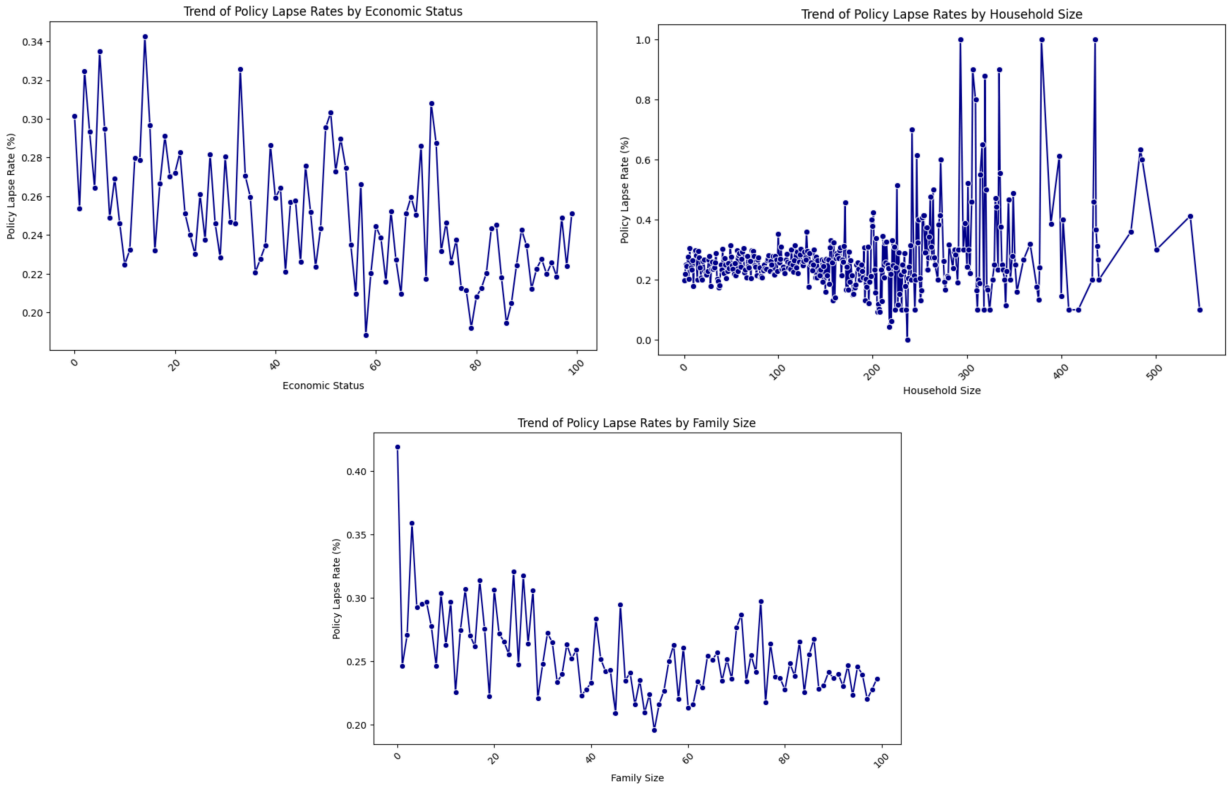
1. Average Product Conversion Rates

In the first graph, which visualizes the average conversion rates across products, we can observe that Product 8 stands out as extremely popular, with a significantly higher conversion rate compared to the other products. On the other hand, Products 0, 3, and 5 have notably lower conversion rates, suggesting that these products are less frequently purchased by customers. This information may be valuable when considering product promotions or adjustments in marketing strategies.



2. Agent Tenure vs. Policy Conversions

The second graph explores the relationship between agent tenure and policy conversions. Given that agent tenure spans a wide range, we decided to group agents into 10 bins for clearer visualization. We also removed potential outliers to ensure a more accurate representation of the data. From this analysis, we can infer that agents with a tenure ranging from 26.9 to 51.8 months have the highest conversion rates. This finding could be valuable when recommending agents to clients, as more experienced agents (within this tenure range) seem to be more effective at converting policies.



3. Trend of Policy Lapse Rates Based on Socioeconomic Factors

We aimed to explore the potential socioeconomic factors that may influence policy lapse rates, focusing on three key criteria: economic status, household size, and family size.

(a) Economic Status:

From the graph analyzing economic status and policy lapse rates, we observed a clear trend: higher-tier economic groups tend to have lower policy lapse rates. This suggests that clients in higher economic status groups are less likely to lapse their policies, potentially due to better financial stability or greater reliance on insurance coverage.

(b) Household Size:

The analysis of household size revealed that clients from larger households are more likely to lapse their policies. This could be attributed to the financial burden of maintaining policies for larger households, leading to lapses due to affordability concerns or prioritization of other expenses.

(c) Family Size:

The relationship between family size and policy lapse rates appears to be relatively ambiguous. There isn't a clear upward or downward trend, suggesting that family size alone may not be a strong determinant of policy lapses. Other factors, such as financial priorities or family dynamics, may play a more significant role in this case.

****

Step 8: Implementing The Machine Learning Model

The objective of this step is to develop a Machine Learning model to optimize the recommendation of financial advisors to customers. We explored a supervised Machine Learning classification model using decision trees and random forests.

A Decision Tree and Random Forest Classifier are employed to enhance recommendation accuracy by predicting agent-customer matches. These models:

- Use supervised learning to classify customer-agent pairs based on past interactions.
- Incorporate multiple features, including customer demographics, agent tenure, and historical policy conversions.
- Provide explainable recommendations, allowing insights into which factors contribute most to successful matches.

Step 8.1: Extracting The Relevant Features

In the process of building a machine learning model, one of the most crucial steps is **feature extraction** or **feature selection**, which involves identifying the most relevant variables from the dataset that will be used to train the model. The choice of features directly impacts the performance of the model, as irrelevant or redundant features may introduce noise, making the model less efficient.

In this step, we are preparing the features for the **Decision Tree** algorithm, which is a supervised learning model that can be used for classification tasks. The goal here is to predict the outcome based on the features, and in our case, we are predicting the most suitable agent based on client's needs, which we will define as the success metric of the model.

The important features that are identified as independent variables include **`cltsex, marryd, race_desc_map, economic_status, household_size_grp and`**

`family_size_grp` as these are the most essential client features that will greatly affect the choice of agent.

The column `agntnum` to be used as the response variable. This is because we are predicting the most suitable agent based on the client's needs to be the success metric of the Decision Tree Machine Learning model.

Extracting The Important Features For Model Implementation:

```
# Supervised Learning (Decision Tree & Random Forest)
features = ['cltsex', 'marryd', 'race_desc_map', 'economic_status',
            'household_size_grp', 'family_size_grp']

# Independent Variables - X
X = final_df[features]

# Response Variable - Y
y = final_df['agntnum']
```

****

Step 8.2: Performing The Train-Test-Split

After extracting the relevant features for the Decision Tree model, the next step in the machine learning pipeline is **splitting the data** into **training** and **testing** sets. This is a crucial step in model evaluation, ensuring that the model is trained on one subset of data and tested on a different, unseen subset. This helps assess how well the model generalizes to new, unseen data.

In this step, we will perform a **train-test split** on the dataset, using 80% of the data for training and the remaining 20% for testing. This ensures that the model learns patterns from a large portion of the data while being evaluated on a separate set to avoid overfitting.

In **scikit-learn**, the `train_test_split` method from the `sklearn.model_selection` module is commonly used for splitting the data. A `random_state` value of 42 is set to prevent accuracy values from changing when the code blocks are run multiple times.

Performing The Train-Test Split:

```
# Importing the train_test_split method from sklearn
from sklearn.model_selection import train_test_split

# Performing The Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state = 42)
```

****

Step 8.3: Creating The Decision Tree Model

Once the data has been split into training and testing sets, the next step in the machine learning pipeline is to create and train the **Decision Tree model**. In this step, we will use the `DecisionTreeClassifier` from the `sklearn.tree` module to build a classification model that can predict the target variable based on the features in the dataset.

To begin, we need to import the `DecisionTreeClassifier` from `sklearn.tree`. This class is designed to handle classification tasks, and it automatically builds a decision tree by splitting the dataset at each node based on the features.

The next step is to train the Decision Tree model using the training data. This involves "fitting" the model on the training dataset (`X_train`, `Y_train`). The model will learn the patterns and relationships between the features (`X_train`) and the target variable (`Y_train`).

Once the model has been trained, it's time to use it to make predictions on the **test dataset** (`X_test`). This allows us to evaluate how well the model generalizes to unseen data. The variable `y_pred` will now contain the predicted values of the target variable (i.e., the number of converted policies) based on the test features.

Training, Fitting And Predicting In Decision Tree Model:

```python
# Importing The DecisionTreeClassifier Model from sklearn
from sklearn.tree import DecisionTreeClassifier

# Initializing The Decision Tree Model
dt_model = DecisionTreeClassifier(random_state = 42)

# Fitting The Decision Tree Model On Training Dataset
dt_model.fit(X_train, y_train)

# Predicting The Outcomes On Test Dataset
y_pred = dt_model.predict(X_test)
```

****

Step 8.4: Creating The Confusion Matrix Of Decision Tree Model

A Confusion Matrix is an essential tool for evaluating the performance of classification models. It provides a detailed breakdown of the predictions made by the model by comparing the actual labels (ground truth) to the predicted labels. This comparison gives us more insight into where the model is performing well and where it is making errors.

In the case of the Decision Tree model, we can use the confusion matrix to evaluate how well it classifies the predicted agent based on the features of the clients identified in Step 8.2.1 . To create a confusion matrix, we need to import the **`confusion_matrix`** method from the **`sklearn.metrics`** package.

A confusion matrix generally looks like this:

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

****

Step 8.5: Evaluating The Accuracy Metrics Of The Decision Tree Model

When evaluating a classification model like the Decision Tree, it is important to assess the model's performance across multiple dimensions to understand how well it is

predicting both the positive and negative classes. We will focus on four key metrics commonly used for model evaluation:

1. **Accuracy**
2. **Precision**
3. **Recall**
4. **F1-Score**

These metrics are based on the confusion matrix, which summarizes the model's predictions. Here's a breakdown of these evaluation metrics and how to calculate them.

1) **Accuracy** - Accuracy measures the overall correctness of the model by calculating the proportion of correctly classified instances out of the total number of instances.

```
Accuracy = (TP + TN) / (TP + FP + TN + FN)
```

2) **Precision** - Precision measures how many of the predicted positive cases were actually positive. It is also known as Positive Predictive Value (PPV).

```
Precision = TP / (TP + FP)
```

3) **Recall** - Recall measures how many actual positive cases were correctly predicted.

```
Recall = TP / (TP + FN)
```

4) **F1-Score** - The F1-score is the harmonic mean of precision and recall, providing a balanced metric when there is an uneven class distribution.

```
F1-Score = 2 * ((Precision * Recall) / (Precision + Recall))
```

We can rely on the **`accuracy_score, precision_score, recall_score`** and **`f1_score`** methods from the sklearn.metrics package to perform the computations.

Obtaining The Accuracy Metrics Of Decision Tree Classifier:

```
# Importing The Necessary Metrices To Consider from sklearn
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Calculating The Accuracy, Precision, Recall and F1-Scores of Decision Tree
dt_accuracy = accuracy_score(y_test, y_pred)
dt_precision = precision_score(y_test, y_pred, average = 'weighted', zero_division = 0)
dt_recall = recall_score(y_test, y_pred, average = 'weighted', zero_division = 0)
dt_f1 = f1_score(y_test, y_pred, average = 'weighted')
```

****

Step 8.6: Determining The Runtime Of The Decision Tree Model

While evaluating a Decision Tree model (or any machine learning model), it is important not only to consider the accuracy and performance metrics but also the runtime of the model. The runtime measures how long the model takes to train, make predictions, and evaluate its performance, which is crucial for assessing its efficiency, especially when dealing with large datasets or real-time applications.

What Contributes To The Runtime?

- **Training Time: The time it takes for the model to learn the patterns in the training dataset.**
- **Prediction Time: The time it takes to generate predictions for the test dataset.**
- **Evaluation Time: The time it takes to compute the performance metrics (e.g., accuracy, precision, recall, etc.).**

In some cases, a highly accurate model might take too long to train or make predictions, which could make it unsuitable for applications that require real-time or near-real-time performance. For this Machine Learning Model, we will track the total time taken for the model to train and predict on the dataset. We will exclude the evaluation time.

We can calculate the runtime of the **Decision Tree model** by using the `time` module in Python. First, we import the `time` module, which allows us to record the start and end times of different stages of the model. To measure the **total time (training + prediction time)**, we record the time just before fitting the model and then subtract it from the time immediately after the prediction process.

Tracking The Time Taken For The Decision Tree Model:

```
# Importing Time
import time

# Start timer
start_time = time.time()

# Decision Tree Process In Step 4.2.3
dt_model.fit(X_train, y_train)
y_pred = dt_model.predict(X_test)

# Stop timer
end_time = time.time()

# Time Difference Calculation
time_diff_dt = end_time - start_time
```

****

Step 8.7: Repeat Step 8.2.3 to Step 8.2.6 For Random Forest Model

We will repeat the process, but using a **Random Forest model** instead of a Decision Tree model.

The Random Forest is an ensemble learning method that combines multiple individual decision trees to improve the predictive performance and reduce overfitting. It is widely used for both classification and regression tasks. Each decision tree is trained on a different subset of the data, and the final prediction is made by aggregating the predictions from all the trees.

In the case of classification such as in the case of Decision Trees, the final prediction is typically made by majority voting - the class that is predicted most frequently by the individual trees becomes the final prediction.

We begin by importing the necessary class from the **`sklearn.ensemble`** module. The class we will use for classification tasks is **`RandomForestClassifier`**. We initialize the **`RandomForestClassifier`** with a specific number of decision trees (n_estimators). For this example, we will use the default value of **`n_estimators = 10`**

The rest of the process is similar to the Decision Tree Model implementation.

Implementing The Random Forest Model:

```
# Importing The Random Forest Classifier from sklearn
from sklearn.ensemble import RandomForestClassifier

# Initializing The Random Forest Model
rf_model = RandomForestClassifier(n_estimators = 10, random_state = 42)

# Fitting The Random Forest Model On Training Dataset
rf_model.fit(X_train, y_train)

# Predicting The Outcomes On Test Dataset
rf_pred = rf_model.predict(X_test)
```

Calculating The Accuracy Metrics Of Random Forest Model:

```
# Calculating The Accuracy, Precision, Recall and F1-Scores of Random Forest
rf_accuracy = accuracy_score(y_test, rf_pred)
rf_precision = precision_score(y_test, rf_pred, average = 'weighted', zero_division = 0)
rf_recall = recall_score(y_test, rf_pred, average = 'weighted', zero_division = 0)
rf_f1 = f1_score(y_test, rf_pred, average = 'weighted')
```

Evaluating The Run-Time Of Random Forest Model:

```
# Importing Time
import time

# Start timer
start_time = time.time()

# Random Forest Process In Step 4.2.7
rf_model.fit(X_train, y_train)
rf_pred = rf_model.predict(X_test)

# Stop timer
end_time = time.time()

# Time Difference Calculation
time_diff_rf = end_time - start_time
```

****

Step 8.8: Testing The Random Forest Model

After implementing both the Decision Tree and Random Forest models, the next crucial step is to test the model's predictive capabilities. This will involve defining a function that recommends the most suitable agent based on the characteristics of the clients. These characteristics are represented by the independent variables (features) such as:

- **Client's sex**
- **Marriage status**
- **Race**
- **Economic status**
- **Household size group category**
- **Family size group category**

The Random Forest model, having been trained on these features, will be used to predict the most suitable agent based on the client's features. Given that the Random Forest model was trained with features such as **`agent_age, agent_gender, agent_marital, agent_tenure, cnt_converted, annual_premium_cnvrt`**, we need to format the client's characteristics (the independent variables) to match the input format expected by the trained model.

Defining The Recommendation Function For Best Agent:

```
# Define The Recommendation Function
def recommend_best_agent(customer_features):
    customer_df = pd.DataFrame([customer_features])
    predicted_agent = rf_model.predict(customer_df)
    return predicted_agent[0]
```

Afterwards, we can create our own client characteristics - provide the values of **`agent_age, agent_gender, agent_marital, agent_tenure, cnt_converted`** and **`annual_premium_cnvrt`** in order to test the **`recommend_best_agent`** function properly. We tested two different clients with totally different characteristics.

Testing The Recommend Best Agent Function:

```
# Example Customer Input (2 customers)
customer_one_features = {'cltsex': 1, 'marryd': 1, 'race_desc_map': 0,
                         'economic_status': 50, 'household_size_grp': 2,
                         'family_size_grp': 2}

customer_two_features = {'cltsex': 0, 'marryd': 2, 'race_desc_map': 1,
                         'economic_status': 20, 'household_size_grp': 5,
                         'family_size_grp': 4}

# Finding Their Recommended Agents
recommended_agent_one = recommend_best_agent(customer_one_features)
recommended_agent_two = recommend_best_agent(customer_two_features)
```

****

Step 9: Optimization The Random Forest Model

From Step 6, we realised that the accuracy of the Decision Tree model/Random Forest model is too low as it predicts less than half the most suitable agents correctly. We will stick to the Random Forest Model for the optimization process Hence, in order to improve the efficiency of the model, we can turn to several optimization strategies which include:

1) **Hyperparameter Tuning Via Grid Search CV**
2) **Hyperparameter Tuning Via Randomized Search CV**
3) **Bagging**
4) **Boosting Using XGBoost**

Step 9.1: Hyperparameter Tuning Via GridSearchCV

Machine learning models often have multiple hyperparameters that can significantly affect their performance. Finding the optimal combination of these hyperparameters manually can be time-consuming and inefficient. GridSearchCV, a technique provided by scikit-learn, automates this process by performing an exhaustive search over a grid of predefined hyperparameter values. This report details how GridSearchCV can be used to optimize a Random Forest Classifier model for improved accuracy.

When using a **Random Forest Classifier**, several hyperparameters influence the model's performance. The key hyperparameters considered in this optimization process are:

- **n_estimators**: Defines the number of trees in the Random Forest. A higher number of trees generally improves performance but increases computational cost.
- **max_depth**: Specifies the maximum depth of each decision tree. A deeper tree captures more complex patterns but may overfit.
- **min_samples_split**: The minimum number of samples required to split an internal node. A smaller value allows more splits, increasing complexity and risk of overfitting.

To optimize the **Random Forest Classifier**, we define a grid of hyperparameter values and use **GridSearchCV** to search for the best combination.

Defining The Parameter Grid For Random Forest:

```
# Importing GridSearchCV for sklearn
from sklearn.model_selection import GridSearchCV

# Defining parameter grid for Random Forest
param_grid = {
    'n_estimators': [20, 50, 100, 200],
    'max_depth': [5, 10, 15, 20],
    'min_samples_split': [2, 4, 6, 8]
}
```

After defining the possible parameter values for the `n_estimators`, `max_depth` and `min_samples_split`, we can perform the GridSearchCV using a default `random_state` of 42 with a scoring metric of accuracy. Afterwards, we can fit the model using the `.fit()` method and subsequently obtain the best parameter values using the `.best_params` method. We use a 5-fold cross-validation approach to ensure the model generalizes well and avoids overfitting.

Performing GridSearchCV And Obtaining Best Parameters:

```
# Performing GridSearchCV Using The Parameter Values Defined Above
rf_grid = GridSearchCV(RandomForestClassifier(random_state = 42), param_grid,
                       cv = 5, scoring = 'accuracy')

# Fitting The Model
rf_grid.fit(X_train, y_train)

# Printing Out The Best parameters
print("Best parameters:", rf_grid.best_params_)
```

Once the optimal hyperparameters have been identified through **GridSearchCV**, they can be applied to initialize a new instance of the **RandomForestClassifier** with the best parameter values. This optimized model is then trained using the **training dataset**, ensuring that it leverages the most effective configuration for improved performance. After training, the model is used to make predictions on the **test dataset**, following the same process as the initial, unoptimized model but now utilizing the refined hyperparameter settings to enhance accuracy and generalization.

Fitting And Predicting The Optimized GridSearchCV Model:

```
# Train the optimized model using optimal parameters obtained
rf_model_optimized = RandomForestClassifier(**rf_grid.best_params_, random_state=42)

# Fit the model on the trainig dataset
rf_model_optimized.fit(X_train, y_train)

# Predict the outcomes using the test set
rf_pred_optimized = rf_model_optimized.predict(X_test)
```

Likewise, we can assess the performance of the optimized Random Forest model obtained through GridSearchCV by evaluating key accuracy metrics. This includes computing the `accuracy_score, precision_score, recall_score` and `f1_score` from the sklearn.metrics module. These metrics provide a comprehensive evaluation of the model's effectiveness, ensuring that it not only achieves high overall accuracy but also maintains a balanced performance across different classes.

```python
# Evaluating The Accuracy Metrices Of Random Forest Optimized GridSearcCV Model
rf_accuracy = accuracy_score(y_test, rf_pred_optimized)
rf_precision = precision_score(y_test, rf_pred_optimized, average = 'weighted', zero_division = 0)
rf_recall = recall_score(y_test, rf_pred_optimized, average = 'weighted', zero_division = 0)
rf_f1 = f1_score(y_test, rf_pred_optimized, average = 'weighted')

# Printing The Scores
print("Random Forest GridSearchCV Model Accuracy:", round(rf_accuracy, 4))
print("Random Forest GridSearchCV Model Precision:", round(rf_precision, 4))
print("Random Forest GridSearchCV Model Recall:", round(rf_recall, 4))
print("Random Forest GridSearchCV Model F1-Score:", round(rf_f1, 4))
```

****

Step 9.2: Hyperparameter Tuning Via RandomizedSearchCV

Hyperparameter tuning plays a crucial role in improving the performance of machine learning models. While GridSearchCV searches exhaustively over all possible hyperparameter combinations, RandomizedSearchCV provides a more efficient approach by randomly sampling from a predefined distribution of hyperparameters. This method significantly reduces computational cost while still effectively identifying optimal parameter values. This report details how RandomizedSearchCV can be used to optimize a Random Forest Classifier model.

RandomizedSearchCV is a hyperparameter tuning method in **scikit-learn** that randomly selects a subset of hyperparameter combinations from a specified range or distribution. The model is then trained and evaluated using **cross-validation**, and the best-performing hyperparameter combination is chosen based on a specified metric (e.g., accuracy, F1-score, precision, recall).

Table Containing Differences Between GridSearchCV And RandomizedSearchCV:

Feature	GridSearchCV	RandomizedSearchCV
Search Method	Exhaustive (Tests all combinations)	Random Sampling from hyperparameter space
Computational Cost	High (exponential growth with parameters)	Low (only a subset is tested)
Most Suitable For	Smaller Parameter Grids	Larger Parameter Space

To optimize the **Random Forest Classifier**, we define a distribution of hyperparameter values and use **RandomizedSearchCV** to search for the best combination. Instead of specifying a fixed set of values, we define a range using **randint** for numerical parameters.

Defining The Parameter Distribution For RandomizedSearchCV:

```
# Importing RandomizedSearchCV fom sklearn
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Define parameter distribution for RandomizedSearchCV
param_dist_r = {
    'n_estimators': randint(50, 500),
    'max_depth': randint(3, 20),
    'min_samples_split': randint(2, 10)
}
```

We use a 5-fold cross-validation approach and set `n_iter=10` to limit the number of random combinations tested.

Performing RandomizedSearchCV And Obtaining Best Parameters:

```
# Perform RandomizedSearchCV
random_search = RandomizedSearchCV(RandomForestClassifier(random_state = 42),
                                   param_dist_r, n_iter = 20, cv = 5,
                                   scoring = 'accuracy')

# Fitting The Model
random_search.fit(X_train, y_train)

# Printing Out The Best parameters
print("Best Parameters:", random_search.best_params_)
```

****

Step 9.3: Bagging

Bagging, short for Bootstrap Aggregating, is an ensemble technique designed to enhance the performance and robustness of machine learning models. By creating multiple models from random subsets of the data, bagging reduces model variance and

prevents overfitting. In the case of the Random Forest algorithm, bagging plays a key role in building diverse decision trees, which are then combined to make predictions that tend to generalize better than individual models.

Bagging Process Overview

The bagging process involves the following key steps:

1. **Data Subsets**: We will generate a number of random subsets (with replacement) from the dataset. Each subset will be used to train a separate Random Forest model.
2. **Random Forest Model**: For each subset, we will create a **RandomForestClassifier** model with a specified number of estimators (trees). The model will be trained on the random subset and then tested on the remaining data.
3. **Model Prediction**: The model will generate predictions based on the test data, and accuracy and other metrics (precision, recall, F1-score) will be calculated for each iteration.
4. **Accuracy Calculation**: We will track the accuracy and other performance metrics for each iteration, and then compute the average accuracy over all iterations.

We first set the number of bagging iterations (`n_iterations`) and initialize lists to store accuracy and performance metrics.

Bagging Set-up Process:

```
# Set the deafult number of iterations of the bagging process
n_iterations = 10

# Create lists for the accuracy metrices
accuracies_bagging = []
precision_bagging = []
recall_bagging = []
f1_score_bagging = []
```

We randomly sample a subset of the dataset with replacement. The number of data points chosen (`m`) is the same as the size of the original dataset. A `RandomForestClassifier` model is then trained on the subset. The trained model makes predictions on the test set. We calculate the accuracy and other performance metrics (precision, recall, F1-score) for the current iteration and store them in the lists defined above.

The accuracy of the Random Forest model might differ depending on two factors, the number of iterations **`n_iterations`** of the Bagging process, as well as the value of the number of data rows that are selected for each iteration of the Bagging process **`n_estimators`**. We set the **`n_iterations`** and **`n_estimators`** to default values of 10 and 50 respectively.

Entire Bagging Process - Fitting, Predicting, Evaluating:

```
# Perform bagging technique multiple times
for i in range(n_iterations):

    # Initiate RandomForestClassifier Model
    bagging_model = RandomForestClassifier(n_estimators = 50)

    # Fit And Train Model, Predict Afterwards
    bagging_model.fit(X_train, y_train)
    y_pred = bagging_model.predict(X_test)

    # Evaluating The Accuracy Metrices
    model_accuracy = accuracy_score(y_test, y_pred)
    model_precision = precision_score(y_test, y_pred, average = 'weighted', zero_division = 0)
    model_recall = recall_score(y_test, y_pred, average = 'weighted', zero_division = 0)
    model_f1 = f1_score(y_test, y_pred, average = 'weighted')

    # Appending The Values Of Accuracy Metrices Into The Respective Lists
    accuracies_bagging.append(model_accuracy)
    precision_bagging.append(model_precision)
    recall_bagging.append(model_recall)
    f1_score_bagging.append(model_f1)
```

****

Step 9.4: Boosting Using XGBoost

XGBoost (Extreme Gradient Boosting) is an advanced machine learning algorithm based on the gradient boosting technique. It is one of the most widely used and powerful models for both classification and regression tasks. The strength of XGBoost lies in its ability to build a strong predictive model by combining several weak learners (usually decision trees), where each new tree attempts to correct the errors made by the previous ones.

Gradient Boosting works by iteratively adding trees to a model, where each tree is trained to reduce the residual errors of the previous trees. XGBoost incorporates additional features like **L1 (Lasso)** and **L2 (Ridge)** regularization, which helps prevent overfitting—a critical improvement over traditional gradient boosting methods.

The effectiveness of XGBoost is influenced by several hyperparameters that control the learning rate, model complexity, and overfitting. The two most important ones that we will focus on are:

1. **learning_rate**:
    - This controls the step size in the gradient descent process when optimizing the model. A lower learning rate makes the model more conservative, leading to a slower but more stable convergence. A higher learning rate speeds up learning but increases the risk of overshooting the optimal solution.

2. **max_depth**:
    - This parameter controls the maximum depth of the decision trees. Deeper trees have more capacity to model complex patterns in the data, but they may also lead to overfitting if the model becomes too complex.

Default Settings:

- We will set the **learning_rate** to **0.1** and the **max_depth** to **10**, which are common default values in many scenarios for a balanced trade-off between model complexity and training time.

In the following implementation, we will perform boosting with XGBoost by iterating over several bagging iterations, similar to the process we followed for Random Forest bagging. However, instead of fitting a random forest model, we will fit an XGBoost model in each iteration. We will initialize the **`XGBClassifier`** from the **`xgboost`** library and set the **learning_rate** and **max_depth** hyperparameters.

XGB Classifier Initialization:

```
# Importing XGBClassifier from xgboost
from xgboost import XGBClassifier

# Initialize The XGBClassifier Model
xgb_model = XGBClassifier(n_estimators = 100, learning_rate = 0.1,
                          max_depth = 10, random_state = 42)
```

Fitting And Predicting Process For XGBoost Model:

```
# Fitting the model on the training dataset
xgb_model.fit(X_train, y_train)

# Predicting the outcomes using the test set
xgb_pred = xgb_model.predict(X_test)
```

Evaluating The Accuracy Metrics Of XGBoost Model:

```
# Determining the accuracy metrics for the XGBClassifier
xgb_accuracy = accuracy_score(y_test, xgb_pred)
xgb_precision = precision_score(y_test, xgb_pred, average = 'weighted', zero_division = 0)
xgb_recall = recall_score(y_test, xgb_pred, average = 'weighted', zero_division = 0)
xgb_f1 = f1_score(y_test, xgb_pred, average = 'weighted')

# Printing The Scores
print("XGBoost Model Accuracy:", xgb_accuracy)
print("XGBoost Model Precision:", xgb_precision)
print("XGBoost Model Recall:", xgb_recall)
print("XGBoost Model F1-Score:", xgb_f1)
```

****

Results:

Type Of Model Used	Accuracy	Precision	Recall	F1-Score	Runtime (s)
Decision Tree	~ 0.18	~ 0.15	~ 0.18	~ 0.15	1.87
Random Forest (Unoptimised)	~ 0.22	~ 0.20	~ 0.22	~ 0.19	18.20
Random Forest (GridSearchCV)	~ 0.32	~ 0.31	~ 0.32	~ 0.31	Very Long
Random Forest (RandomizedSearchCV)	~ 0.28	~ 0.26	~ 0.28	~ 0.26	Long
Random Forest (Bagging)	~ 0.25	~ 0.23	~ 0.25	~ 0.22	Very Long
Random Forest (XGBoost)	~ 0.60	~ 0.55	~ 0.60	~ 0.53	Extremely Long

1. Decision Tree

- Performance: The Decision Tree model shows the lowest accuracy (~18%) with poor precision (~15%) and recall (~18%).
- Runtime: Very fast (1.87s).
- Insight: A simple Decision Tree is not sophisticated enough for this problem, likely leading to overfitting and poor generalization.

2. Random Forest (Unoptimized)

- Performance: Shows slight improvement over the Decision Tree but still performs poorly (~22% accuracy).
- Runtime: Significantly longer (18.2s).
- Insight: While ensemble learning helps, an unoptimized Random Forest does not yield significant performance gains.

3. Random Forest (GridSearchCV)

- Performance: Improved performance (~32% accuracy, ~31% precision, ~32% recall).
- Runtime: Very long (occasional memory crashes)
- Insight: GridSearchCV helps find better hyperparameters but at a high computational cost.

4. Random Forest (RandomizedSearchCV)

- Performance: Slightly worse than GridSearchCV (~28% accuracy).
- Runtime: Still very long (but faster than GridSearchCV) (occasional memory crashes)
- Insight: Randomized search provides a quicker but less exhaustive parameter search compared to GridSearchCV.

5. Random Forest (Bagging)

- Performance: Moderate improvement over the unoptimized version (~25% accuracy).
- Runtime: Still very long (occasional memory crashes)
- Insight: Bagging reduces variance but does not drastically improve accuracy.

6. Random Forest (XGBoost)

- Performance: The best model with ~60% accuracy, significantly outperforming others.
- Runtime: Extremely long (frequent memory crashes)
- Insight: XGBoost boosting technique and optimization make it superior, indicating that gradient boosting is the best choice for this problem.

****

Insights:

(a) Decision Tree (~18% Accuracy, Fastest Runtime - 1.87s)

- Reason for Low Accuracy:
    - A single decision tree is a weak learner, prone to overfitting because it tries to perfectly classify training data rather than generalizing to unseen data.
    - It does not incorporate ensemble learning, making it highly sensitive to noise in training data.
- Why is the Runtime Fast?
    - A decision tree makes greedy, localized decisions at each node, reducing computational complexity.
    - The tree-building process is quick as it only involves splitting features at various thresholds without requiring iterative updates.

(b) Random Forest (Unoptimized) (~22% Accuracy, 18.2s Runtime)

- Why Accuracy Improves?
    - Random Forest is an ensemble of multiple decision trees, reducing overfitting by averaging predictions.
    - The introduction of random feature selection improves generalization compared to a single decision tree.
- Why Runtime Increases?
    - Instead of training one tree, the model builds multiple trees (default: 100), each requiring separate training runs.
    - Predictions require majority voting across trees, making inference slower than a single decision tree.

(c) Random Forest (GridSearchCV) (~32% Accuracy, Very Long Runtime)

- Why Accuracy Increases?

    - Hyperparameter tuning selects the best combination of parameters (e.g., tree depth, number of trees, feature splits).
    - This leads to better generalization, boosting accuracy.
- Why Runtime Becomes Very Long?
    - Exhaustive Search: GridSearchCV evaluates all possible parameter combinations across multiple cross-validation splits.
    - If we have `n` hyperparameters with `m` possible values each, the complexity is O(m^n), making it computationally expensive.

(d) Random Forest (RandomizedSearchCV) (~28% Accuracy, Very Long Runtime)

- Why is it Slightly Worse than GridSearchCV?
    - Instead of searching all combinations, it randomly samples parameter sets.
    - While it can find good parameters, it may miss the optimal combination due to random selection.
- Why is Runtime Still High?
    - It still requires training multiple models but is typically faster than GridSearchCV because it searches fewer combinations.

(e) Random Forest (Bagging) (~25% Accuracy, Very Long Runtime)

- Why Accuracy is Lower than GridSearch?
    - Bagging reduces variance by averaging multiple models but does not inherently improve individual tree performance.
    - Unlike boosting (e.g., XGBoost), it does not correct errors sequentially.
- Why is Runtime High?
    - Like a standard Random Forest, bagging involves training multiple decision trees, which is computationally expensive.

(f) XGBoost (Best Model - ~60% Accuracy, Very Long Runtime)

- Why Does it Perform Best?
    - Gradient Boosting Mechanism: Instead of building trees independently, XGBoost builds them sequentially, with each tree correcting errors made by the previous ones.
    - Regularization: XGBoost includes L1 and L2 regularization, preventing overfitting.
    - Feature Importance Handling: Automatically assigns higher importance to influential features.

- Why is Runtime Long?
    - Boosting requires iterative updates, making training computationally expensive.
    - It needs multiple passes over the data to update weak learners, increasing the time required.

Accuracy-Runtime Tradeoff

Understanding the Tradeoff

1. Simple models (Decision Tree) run faster but perform worse because they do not leverage ensemble learning.
2. Ensemble methods (Random Forest, Bagging) improve accuracy but take longer due to multiple tree training.
3. Hyperparameter tuning (GridSearchCV, RandomizedSearchCV) increases accuracy further but adds computational cost.
4. XGBoost achieves the best accuracy but requires the most computation, as it builds trees sequentially.

Most Suitable Model & Justification

Best Choice: XGBoost

Reasoning:

1. Best Accuracy (60%):
    - The large performance gap suggests that boosting mechanisms are crucial for this task.
2. Handles Complex Patterns:
    - XGBoost's ability to correct previous errors sequentially allows it to learn more complex relationships than Random Forest.
3. Regularization Prevents Overfitting:
    - Unlike Decision Trees and Random Forest, XGBoost incorporates L1/L2 penalties, making it more robust.
4. Efficient Handling of Missing Values:
    - XGBoost automatically learns how to deal with missing data, whereas Random Forest requires imputation strategies.

Potential Downsides

- Long Training Time: If computational cost is a concern, XGBoost may require GPU acceleration or hyperparameter tuning to speed up training.
- Harder to Interpret: XGBoost models are complex and require additional methods (e.g., SHAP values) for explainability.

****

Ethical Considerations When Handling The Dataset:

When training a machine learning model using decision trees to predict the most suitable financial advisor (agent) for a client, there are several critical ethical considerations that must be taken into account to ensure fairness, privacy, and transparency.

Below is a detailed discussion of these considerations:

1. Avoiding Discriminatory Practices

- A key concern when building predictive models is the potential for inadvertently embedding existing biases from historical data. If the model is trained on data that reflects biases—whether related to race, gender, age, or marital status—the model's predictions may perpetuate these biases. This could lead to unfair treatment of certain customer groups, ultimately resulting in less optimal matches between customers and advisors.
- To avoid such discrimination, it is important to assess the dataset for any existing biases and ensure that the model does not reinforce these biases in its recommendations. Careful examination of historical data and techniques such as **reweighting** or **rebalancing** the dataset can help mitigate this risk. Furthermore, **fairness-aware** algorithms can be employed to prevent biased outcomes.

2. Ethical Models

- It is essential that the model does not discriminate against customers or financial advisors based on sensitive characteristics, such as gender, race, or marital status. Bias detection techniques and fairness constraints can be implemented during the model development process to ensure that predictions are equitable and not influenced by inappropriate factors.
- Methods like **adversarial debiasing**, **counterfactual fairness**, and **fairness constraints** can be integrated into the decision tree algorithm to reduce the

likelihood of bias, ensuring that the model makes decisions based on relevant criteria such as expertise, experience, and success rate rather than irrelevant demographic factors.

3. Transparency

- Decision trees are generally considered interpretable models, which is a crucial advantage. However, it is important to maintain **transparency** in how the model makes its decisions. Stakeholders, including customers, financial advisors, and regulatory bodies, should be able to understand the rationale behind the model's recommendations.
- Providing clear explanations of how certain attributes (e.g., customer age, product preferences, or advisor expertise) affect the model's predictions will help maintain trust and mitigate the potential for mistrust or misinterpretation. A model that can offer clear, interpretable results helps promote accountability and informed decision-making.

4. Handling Missing or Incomplete Data

- In real-world datasets, it is common for some data points to be missing or incomplete (e.g., missing values for a customer's age, marital status, or product group). Ethical considerations should guide how these missing values are handled to ensure that the model does not make decisions based on inaccurate assumptions or distorted data.
- It is important to employ ethical imputation methods—such as using domain knowledge, statistical imputation, or prediction models—to replace missing values in a manner that does not unfairly distort the model's predictions. Additionally, transparency should be maintained regarding how missing data is handled, so that all stakeholders understand the potential limitations of the model.

5. Data Adjustments

- The dataset in this case has been adjusted to ensure privacy and security, with modifications made to numerical values and product names being masked. While these changes are necessary for data protection, it is essential to ensure that these modifications do not inadvertently impact the performance or accuracy of the model.
- Any adjustments made to the data should be carefully considered to ensure that they do not introduce bias or obscure important patterns in the data. Transparency in the data preparation process is crucial for ensuring that stakeholders trust the final model's predictions.

6. Informed Consent

- Customers whose data is used to train the model should have provided **informed consent**, fully understanding how their data will be utilized in the model-building process. This is important for ensuring that customers are aware of how their financial information will be used to match them with advisors.
- Proper consent mechanisms should be implemented to ensure that customers have the opportunity to review how their data will be used, with clear explanations provided regarding any potential risks or benefits of participation.

7. Model Accountability

- One of the key advantages of using decision trees over other more complex models is that they are relatively easy to interpret. This transparency is valuable for accountability, as it allows customers and stakeholders to request an explanation for the model's decisions.
- It is crucial that customers feel confident in the recommendations provided by the model and that they are able to contest any decisions that they feel are inappropriate. Models that can offer clear, explainable results provide greater accountability compared to "black-box" models, ensuring that all stakeholders can trust the model's outcomes and engage in meaningful dialogue regarding any concerns or adjustments.

****

Conclusion:

Through this project, we successfully leveraged **Decision Tree and Random Forest models** to enhance financial advisor-client matching and predict policy conversions. By applying a structured **data-driven approach**, we tackled key challenges related to data preprocessing, feature engineering, and model optimization, ultimately improving the accuracy and reliability of our predictions.

The **data cleaning and transformation steps** were essential in ensuring the quality of our input features. Handling missing values, encoding categorical variables, and scaling numerical attributes helped standardize the dataset for better model performance. The **Decision Tree model** provided interpretable results, making it easier to understand the factors influencing policy conversions, while the **Random Forest model** demonstrated strong predictive power by reducing overfitting through ensemble learning.

One of the biggest challenges faced was dealing with **class imbalance and sparse data**, which affected model training and generalization. To overcome this, we carefully tuned model parameters and optimized feature selection to ensure **fair representation** of different customer and agent groups. Additionally, feature importance analysis provided **actionable insights** into the most significant predictors of successful policy purchases, which can guide business strategies for Singlife in refining its advisor-client allocation process.

By implementing machine learning techniques, we have demonstrated how data science can enhance decision-making in the **insurance industry**. Our approach provides a scalable framework that can be further **improved and adapted** as more data becomes available. Moving forward, integrating **real-time model updates** and exploring additional models could further refine predictions and enhance customer engagement strategies.

This project not only highlights the feasibility of machine learning for financial advisory services but also underscores the importance of **data quality, feature selection, and model interpretability** in building reliable predictive systems. The combination of **Decision Tree and Random Forest models** has allowed us to balance **accuracy, interpretability, and scalability**, ultimately presenting a robust solution to the problem statement.

****