University of Messina

Machine Learning

(Project Report)

PREDICTING CUSTOMER CHURN FOR A TELECOMMUNICATION COMPANY:

Prof. Giacomo Fiumara
Made by: Shehryar Raja

Table of Contents:

Introduction:

In today's competitive telecom industry, keeping customers has become harder than ever. Customer churn—when customers switch from one service provider to another—can seriously hurt a company's revenue and overall success. For telecom companies, predicting which customers are likely to leave is very important, as it's often much more expensive to gain new customers than to keep the ones they already have. Spotting customers at risk of leaving early allows companies to take steps to keep them, improving customer satisfaction and reducing churn.

Machine learning has become a valuable tool for predicting customer behavior, including churn. By using historical data, machine learning can spot patterns and trends that might not be obvious, leading to better predictions. This project focuses on creating a model that can predict customer churn using various machine learning techniques. We'll analyze and compare models like logistic regression, decision trees, random forests, and SVM to figure out which one works best at predicting churn and reducing its impact on telecom companies.

Understanding the Dataset:

This dataset contains information about telecom customers, with a total of 3,749 records. Each record represents a customer, including details like their age, the services they use, and whether they left the company (churned).

Key columns include:

- **CustomerID**: A unique ID for each customer.
- **Age**: The customer's age (some are missing).
- **Gender**: Whether the customer is male or female.
- **Tenure**: How long the customer has been with the company (in months).
- **Service_Internet, Service_Phone, Service_TV**: What services the customer is using (like internet, phone, or TV).
- **Contract**: What kind of contract the customer has, such as month-to-month or longer-term?
- **Payment Method**: How the customer pays, like by mailed check or online payment.
- **Monthly Charges and Total Charges**: How much the customer pays monthly and in total.
- **Churn**: Whether the customer has left the company ("Yes" or "No").

This dataset will be used to build a model that predicts whether a customer will leave the company (churn). To do this, we will look for patterns in the data, such as which services or payment methods are more common for customers who leave. Some data is missing (like age in some cases), so we may need to clean it up before using it for predictions.

Import all necessary libraries:

1. Loading and Understanding the Dataset

```
[1] # Import necessary libraries
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import seaborn as sns
    from sklearn.preprocessing import LabelEncoder
    from sklearn.model_selection import train_test_split
    from sklearn.linear_model import LogisticRegression
    from sklearn.preprocessing import StandardScaler
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
    from sklearn.model_selection import train_test_split, GridSearchCV
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
    from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, classification_report
    from sklearn.svm import SVC
    from tabulate import tabulate  # To create a table format for the metrics
```

Here we loaded the dataset, and we also displayed the first few rows of the dataset to understand the structure of the dataset.

```
[2] # Load the dataset
    dataset = pd.read_csv('dataset_churn.csv')
```

```
# Display the first few rows
print(dataset.head())
```

```
   Unnamed: 0                            CustomerID   Age  Gender  Tenure  \
0           0  08729464-bde6-43bc-8f63-a357096feab1  56.0    Male      13
1           1  af95bc95-baf4-4318-a21d-70d2ea3148b7  69.0    Male      13
2           2  1fe7eee6-2227-4400-9998-4d993f4a60fd  46.0    Male      60
3           3  f736fe7b-1b44-4acd-84c2-21c4aef648be  32.0  Female      57
4           4  4b40d12d-7633-4309-96b8-aee675ea20ae  60.0    Male      52

  Service_Internet Service_Phone Service_TV        Contract     PaymentMethod  \
0              DSL           Yes         No        One year      Mailed check
1              DSL            No        Yes        Two year      Mailed check
2      Fiber optic            No        Yes  Month-to-month      Mailed check
3      Fiber optic           Yes        Yes  Month-to-month     Bank transfer
4      Fiber optic           Yes        Yes        Two year  Electronic check

   MonthlyCharges  TotalCharges StreamingMovies StreamingMusic OnlineSecurity  \
0           71.88        931.49              No             No            Yes
1          110.99       1448.46             Yes            Yes             No
2          116.74       6997.73             Yes            Yes             No
3           78.16       4452.13              No            Yes             No
4           30.33       1569.73             Yes             No            Yes

  TechSupport Churn
0          No    No
1          No    No
2          No    No
3         Yes    No
4         Yes    No
```

In this step, the code is analyzing the structure and basic statistics of the dataset:

1. **`dataset.info()`**: This shows an overview of the dataset, including the number of entries (3,749), column names, how many non-empty values each column has, and the data types (e.g., integers, objects, or floats).

```
[2]  # Load the dataset
     dataset = pd.read_csv('dataset_churn.csv')

     # Display the first few rows
     print(dataset.head())
```

```
   Unnamed: 0                            CustomerID   Age  Gender  Tenure  \
0           0  08729464-bde6-43bc-8f63-a357096feab1  56.0    Male      13
1           1  af95bc95-baf4-4318-a21d-70d2ea3148b7  69.0    Male      13
2           2  1fe7eee6-2227-4400-9998-4d993f4a60fd  46.0    Male      60
3           3  f736fe7b-1b44-4acd-84c2-21c4aef648be  32.0  Female      57
4           4  4b40d12d-7633-4309-96b8-aee675ea20ae  60.0    Male      52

  Service_Internet Service_Phone Service_TV        Contract     PaymentMethod  \
0              DSL           Yes         No        One year      Mailed check
1              DSL            No        Yes        Two year      Mailed check
2      Fiber optic            No        Yes  Month-to-month      Mailed check
3      Fiber optic           Yes        Yes  Month-to-month     Bank transfer
4      Fiber optic           Yes        Yes        Two year  Electronic check

   MonthlyCharges  TotalCharges StreamingMovies StreamingMusic OnlineSecurity  \
0           71.88        931.49              No             No            Yes
1          110.99       1448.46             Yes            Yes             No
2          116.74       6997.73             Yes            Yes             No
3           78.16       4452.13              No            Yes             No
4           30.33       1569.73             Yes             No            Yes

  TechSupport Churn
0          No    No
1          No    No
2          No    No
3         Yes    No
4         Yes    No
```

2. `dataset.describe()`: This provides basic statistics for the numerical columns, such as:
    - **Count**: How many values are there (non-empty).
    - **Mean**: The average value.
    - **Std (standard deviation)**: How spread out the values are.
    - **Min and Max**: The smallest and largest values.
    - **25%, 50%, 75%**: Percentiles showing the distribution of the data.
    - This step helps understand what the data looks like, what needs cleaning (like missing values), and what the numerical ranges are for different columns.

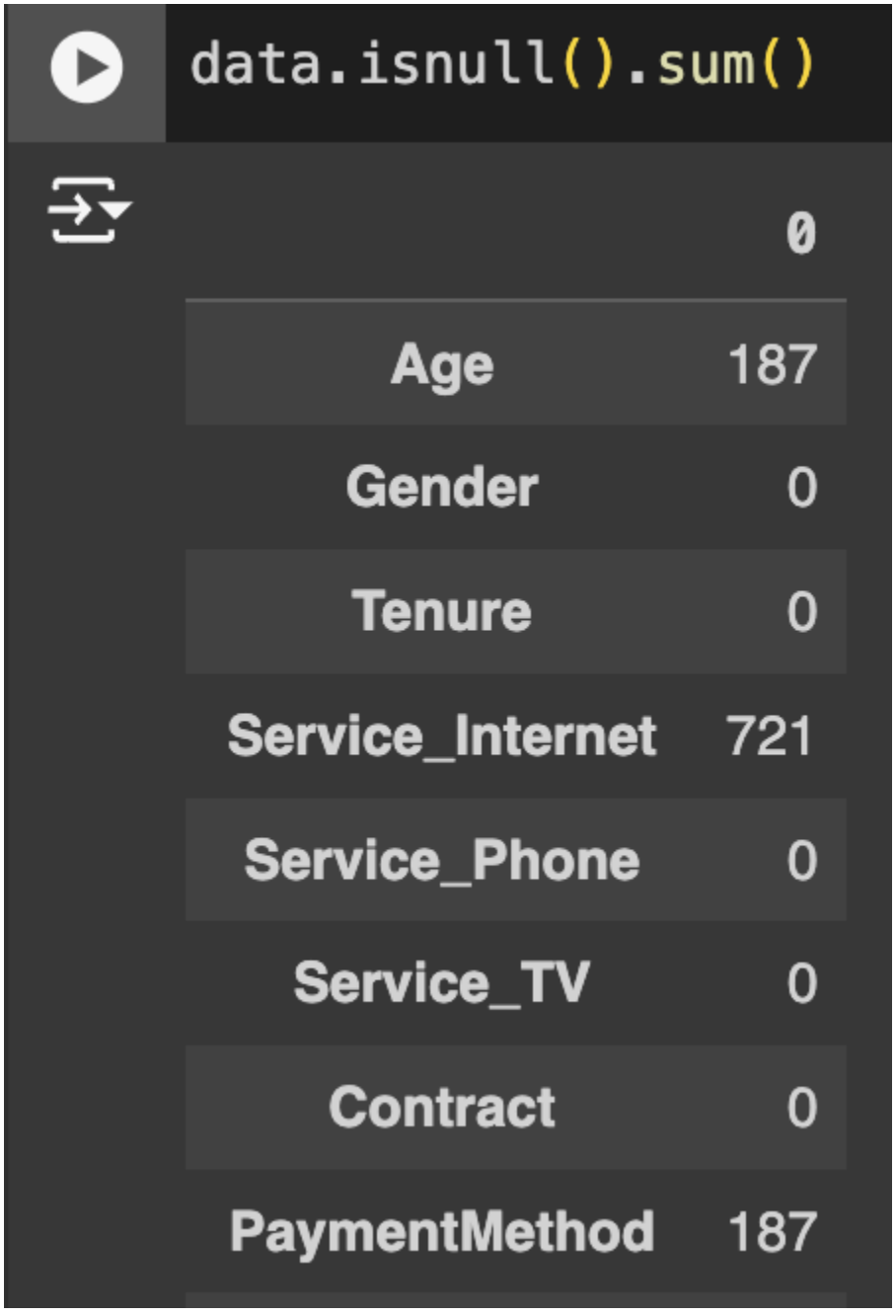
	Unnamed: 0	Age	Tenure	MonthlyCharges	TotalCharges
count	3749.000000	3562.000000	3749.000000	3749.000000	3749.000000
mean	1874.000000	43.655531	36.264070	75.844318	2718.968266
std	1082.387408	14.914474	20.505528	73.062971	3211.879149
min	0.000000	18.000000	1.000000	20.000000	13.190000
25%	937.000000	31.000000	19.000000	44.570000	1076.240000
50%	1874.000000	44.000000	36.000000	69.590000	2132.260000
75%	2811.000000	56.000000	54.000000	95.540000	3619.710000
max	3748.000000	69.000000	71.000000	1179.300000	79951.800000

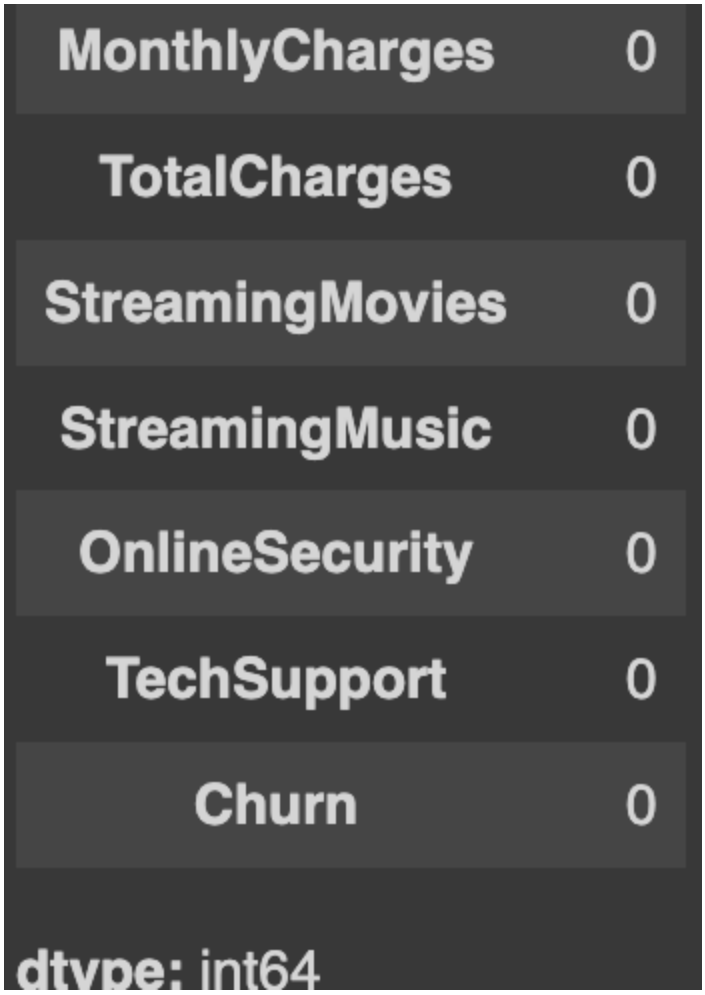
Data Preprocessing:

This code removes two columns, 'Unnamed: 0' and 'CustomerID', from the dataset because they are not useful for the analysis. The drop() function creates a new dataset without these columns, making it cleaner and easier to work with.

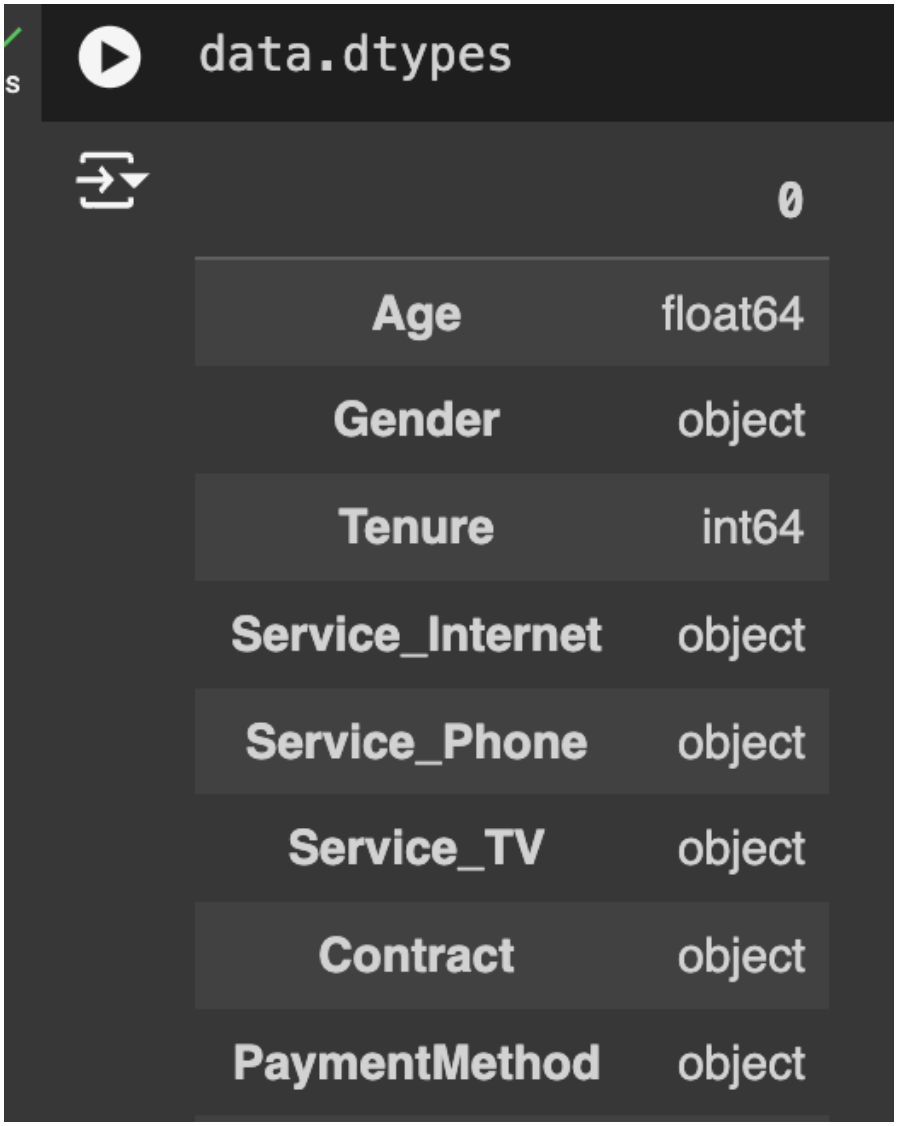
```
0s
# Dropping unnecessary columns
data = dataset.drop(columns=['Unnamed: 0', 'CustomerID'])
```

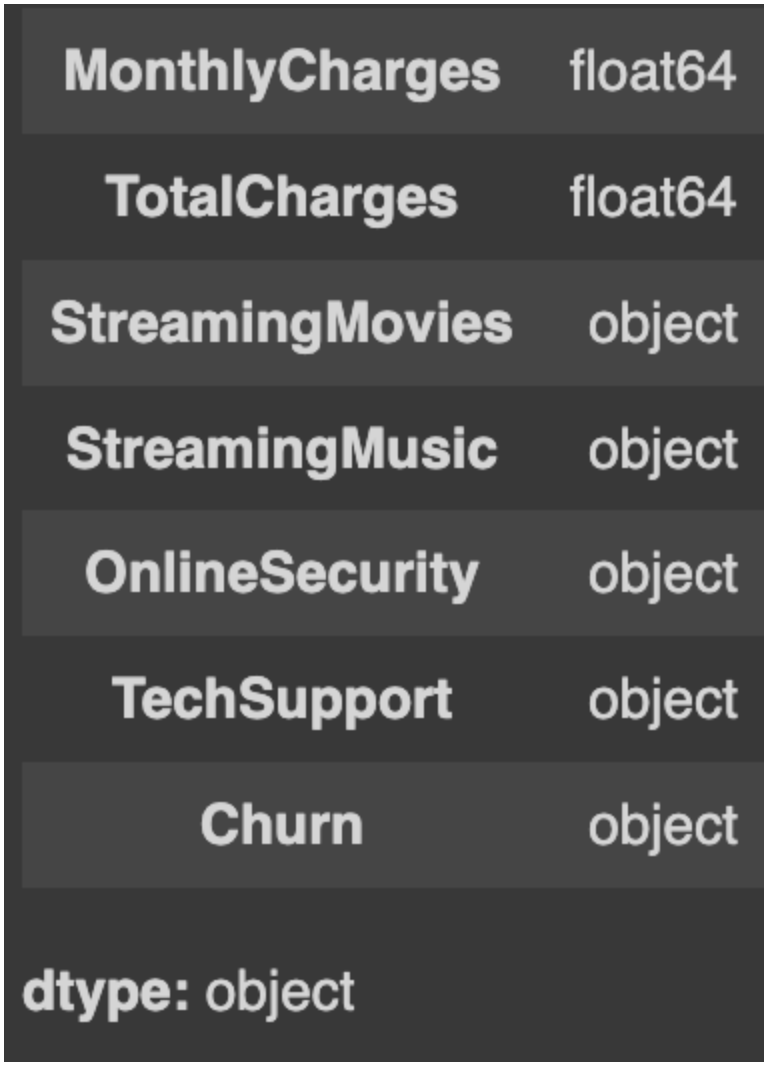
This code checks for any missing (null) values in the dataset. It counts how many missing values there are in each column and displays the result.





This code shows the data types of each column in the dataset. It tells you whether a column contains numbers (like integers or floats) or text (objects).


data.dtypes
0
Age float64
Gender object
Tenure int64
Service_Internet object
Service_Phone object
Service_TV object
Contract object
PaymentMethod object



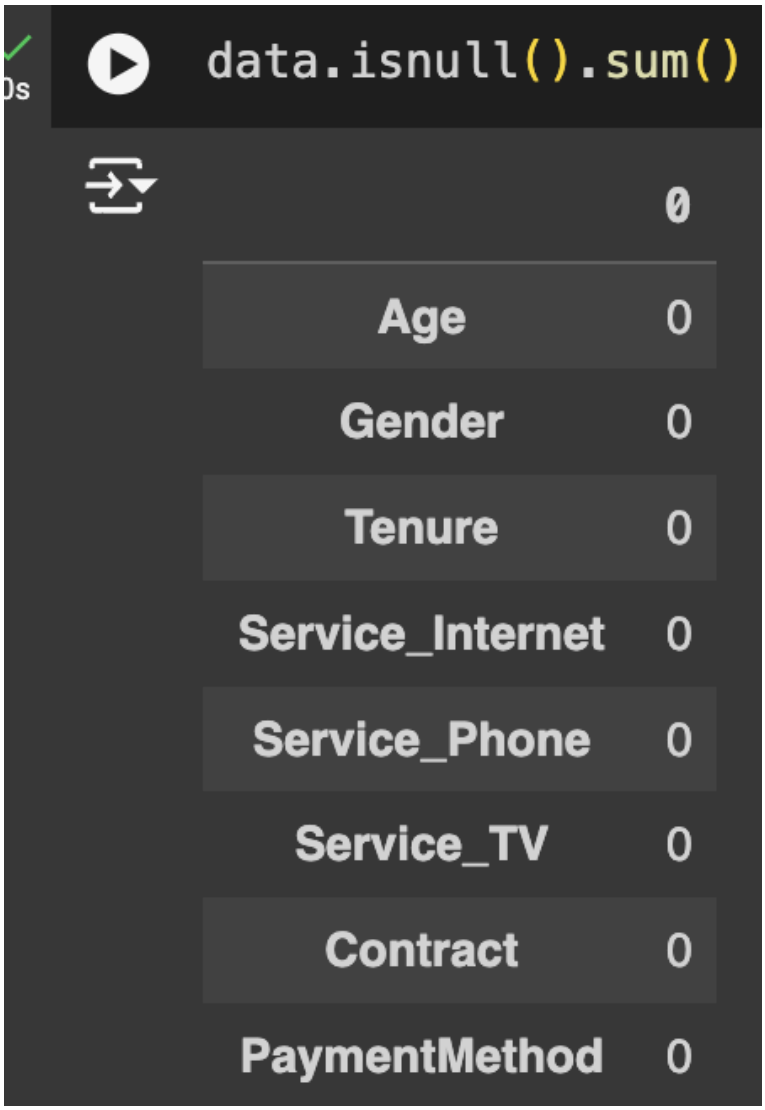
This code fills in missing values in the dataset:

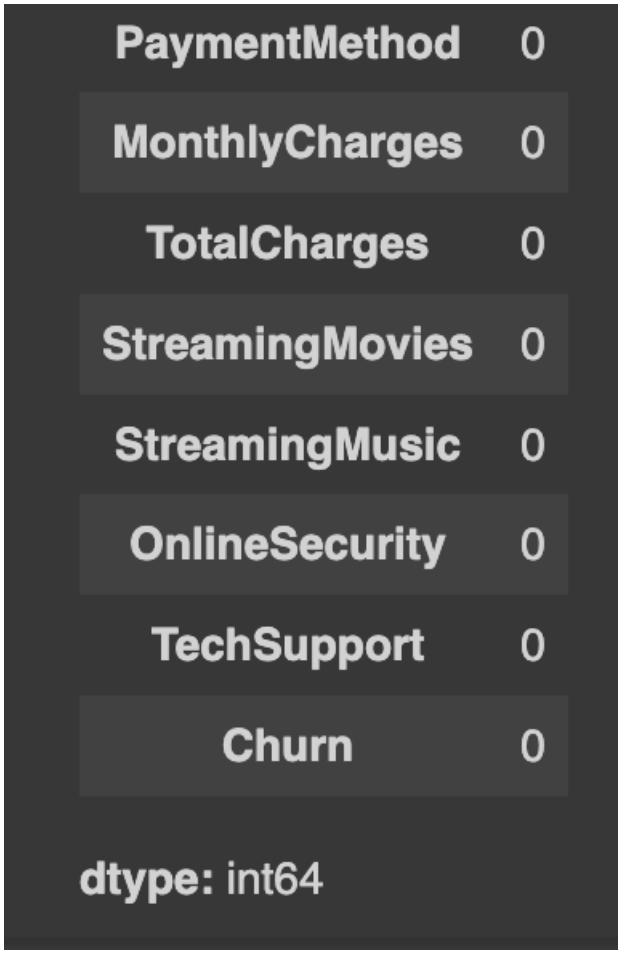
1. **For the 'Age' column**, it replaces missing values with the median age (the middle value) to avoid empty spots.
2. **For the 'PaymentMethod' and 'Service_Internet' columns**: It fills missing values with the most common value (called the "mode") for each of these columns.

This helps clean the data so there are no gaps.

```
[8]  #Handaling the missing value of the age
     data['Age'].fillna(data['Age'].median(), inplace=True)

     #Handaling the missing value of the categorical
     data['PaymentMethod'].fillna(data['PaymentMethod'].mode()[0], inplace=True)

     data['Service_Internet'].fillna(data['Service_Internet'].mode()[0], inplace=True)
```

This code counts the number of missing (null) values in each column of the dataset and shows the total for each. It's used to quickly check where data is missing.





This code changes text data into numbers:

1. **Pick columns** with text (like 'Gender', 'Churn').
2. **Label encoding**: Turns text values into numbers (e.g., "male" becomes 0, "female" becomes 1).
3. **Loop**: Applies this change to all selected columns.

At the end, it shows the updated dataset, where the text has been replaced by numbers.

```python
# List of categorical columns to be encoded
category_cols = ['Gender', 'Service_Internet', 'Service_Phone', 'Service_TV',
                 'Contract', 'PaymentMethod', 'StreamingMovies', 'StreamingMusic',
                 'OnlineSecurity', 'TechSupport', 'Churn']

# Applying Label Encoding to each categorical column
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

for col in category_cols:
    data[col] = encoder.fit_transform(data[col])

# Display the first few rows of the updated dataset
data.head()
```

	Age	Gender	Tenure	Service_Internet	Service_Phone	Service_TV	Contract	Payme
0	56.0	1	13	0	1	0	1	
1	69.0	1	13	0	0	1	2	
2	46.0	1	60	1	0	1	0	
3	32.0	0	57	1	1	1	0	
4	60.0	1	52	1	1	1	2	

This code shows boxplots for four columns to help understand the data:

1. **Pick columns**: The list includes columns like 'Age', 'Tenure', etc.
2. **Draw boxplots**: A boxplot shows the range of values and highlights any extreme values (outliers).
3. **Arrange plots**: Makes sure the plots are neatly arranged.
4. **Show plots**: Displays the boxplots for all the chosen columns.

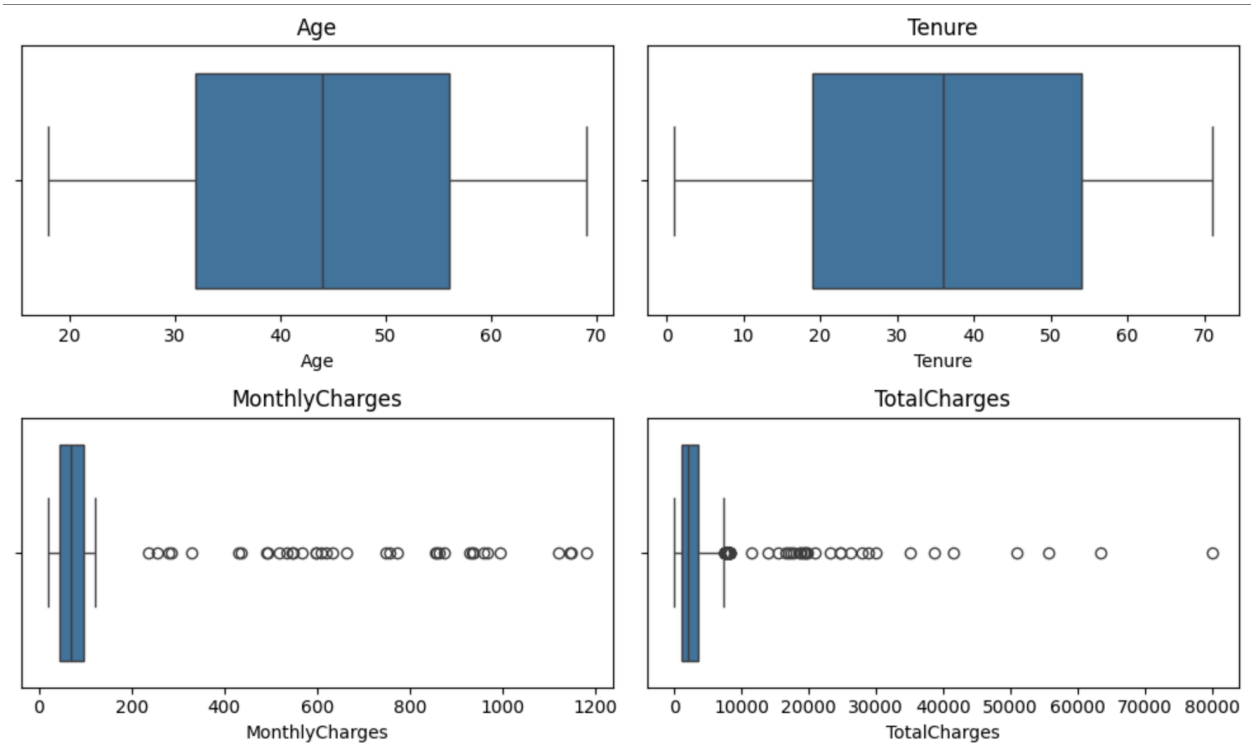
In short, it helps you see how the data is spread out and spot any unusual values.

```python
# Ensure 'numerical_feature' is defined and contains correct column names
# For example:
numerical_feature = ['Age','Tenure', 'MonthlyCharges', 'TotalCharges']
plt.figure(figsize=(10, 6))
for i, feature in enumerate(numerical_feature, 1):
    plt.subplot(2, 2, i)
    sns.boxplot(x=data[feature])  # Use 'feature' to access column
    plt.title(feature)

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

This box plot shows the distribution of four features: **Age**, **Tenure**, **MonthlyCharges**, and **TotalCharges**.

- **Age** and **Tenure** look balanced without extreme values (outliers).
- **MonthlyCharges** and **TotalCharges** have many outliers, shown as circles outside the main box.

The box represents the middle 50% of data (between the 1st and 3rd quartiles), and the line inside is the median. The lines (whiskers) show the range of most data, and anything outside of this is considered an outlier.



This code replaces extreme values (outliers) in your data with the median value of that column.

- It checks if the column exists and if it contains numbers.
- It calculates limits to identify outliers (values too high or too low).
- Any values outside these limits are replaced by the median (middle value) of the column.
- Finally, it applies this process to all numerical columns in your dataset.

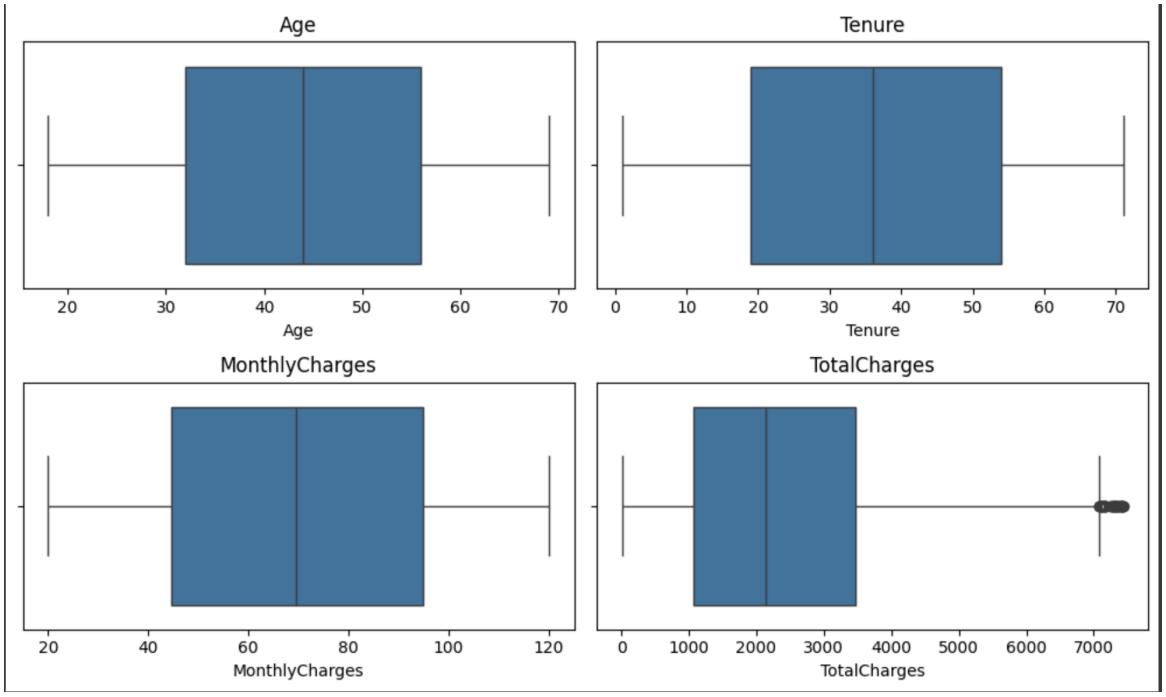
```
[12] def replace_outliers_with_median(data, column):
         # Check if column exists in the DataFrame
         if column not in data.columns:
             print(f"Column '{column}' not found in the DataFrame!")
             return data

         # Ensure the column is numeric
         if not np.issubdtype(data[column].dtype, np.number):
             print(f"Column '{column}' is not numeric, skipping...")
             return data

         # Calculate the first and third quartiles
         Q1 = data[column].quantile(0.25)
         Q3 = data[column].quantile(0.75)
         IQR = Q3 - Q1

         # Determine the lower and upper bounds for outliers
         lower_bound = Q1 - 1.5 * IQR
         upper_bound = Q3 + 1.5 * IQR

         # Calculate the median value for the column
         median_value = data[column].median()

         # Replace outliers with the median
         data[column] = np.where(
             (data[column] < lower_bound) | (data[column] > upper_bound),
             median_value,
             data[column]
         )

         return data

     # Loop over the numerical features and apply the outlier replacement
     # Use 'numerical_feature' instead of 'numerical_features'
     for feature in numerical_feature:
         data = replace_outliers_with_median(data, feature)
```

This code creates boxplots for four numerical features from our dataset, displaying them in a 2x2 grid. It helps visualize the distribution and outliers for each feature.

```
[13] plt.figure(figsize=(10, 6))
     # Iterate over the correct variable name 'numerical_feature'
     for i, feature in enumerate(numerical_feature, 1):
         plt.subplot(2, 2, i)
         sns.boxplot(x=data[feature])
         plt.title(feature)
     plt.tight_layout()
     plt.show()
```

This image shows boxplots for four columns: **age**, **tenure**, **monthly charges**, and **total charges**.

1. **Age**: Most customers are between 30 and 60 years old. No outliers.
2. **Tenure**: Customers have been with the company from 1 to 71 months. No outliers.
3. **Monthly Charges**: Charges range from $20 to $120 per month. No outliers.
4. **TotalCharges**: Charges range from $0 to over $7,000. There are some high outliers (dots beyond the box).



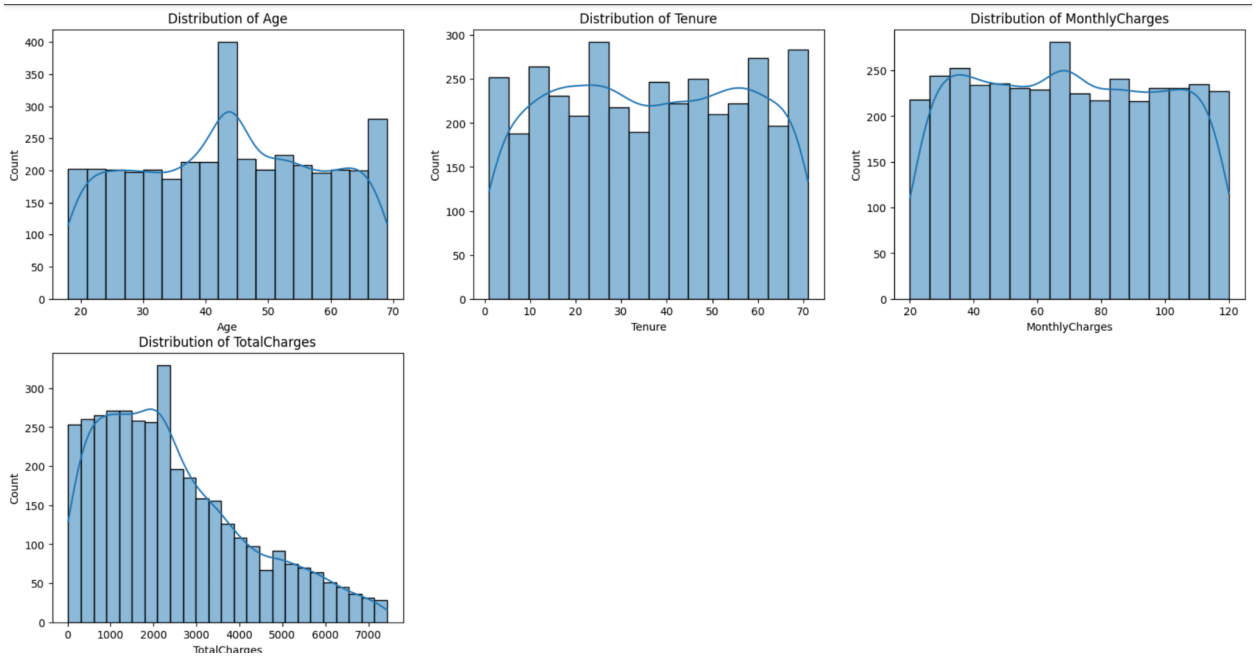
EDA(Exploratory DataAnalysis):

This code shows **histograms** for each numerical feature to visualize how the data is distributed, with smooth curves (KDE) added. It creates multiple plots in a grid format.

```
plt.figure(figsize=(20,10))
for i , col in enumerate(numerical_feature, 1):
    plt.subplot(2,3,i)
    sns.histplot(data[col],kde=True)
    plt.title(f'Distribution of {col}')
plt.show()
```

The image shows four charts representing different customer characteristics:

1. **Age**: Most customers are between 40-50 years old.
2. **Tenure**: The length of time customers have been with the service is spread evenly, from new to long-term users.
3. **Monthly Charges**: Charges vary, with most customers paying around $60 per month.
4. **Total Charges**: Most customers have lower total costs, with fewer having very high total amounts.

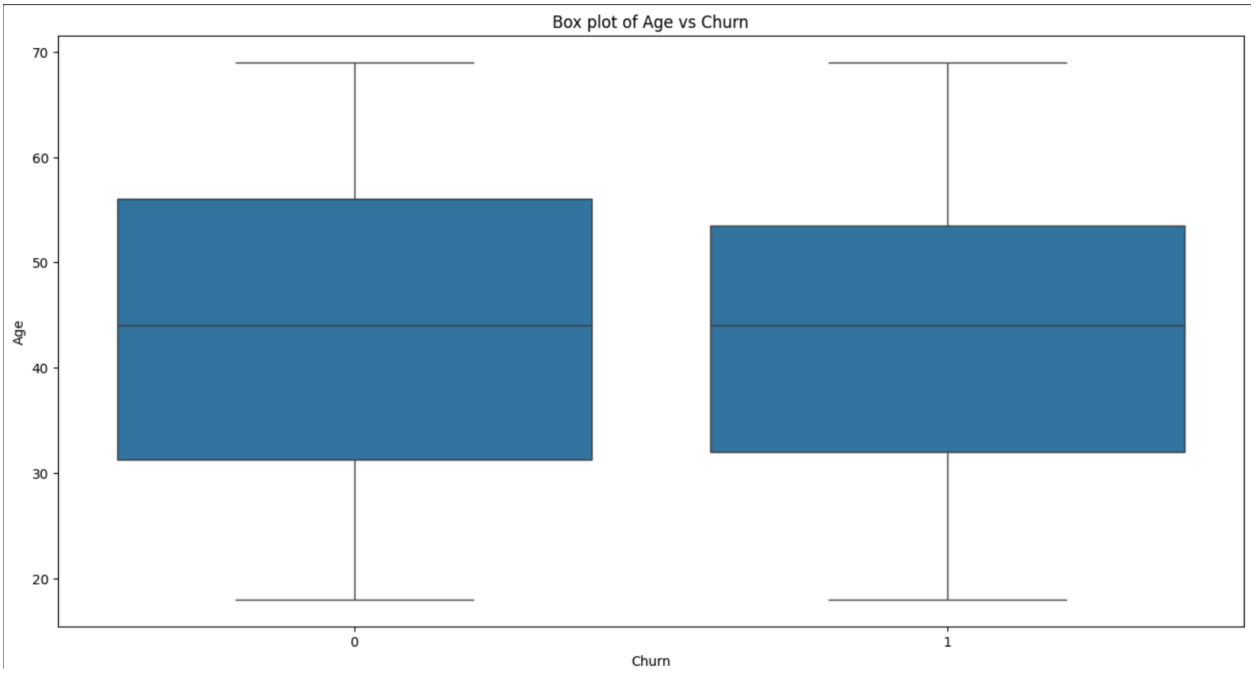


This code creates box plots to visualize the distribution of numerical features against a target variable, `churn`.

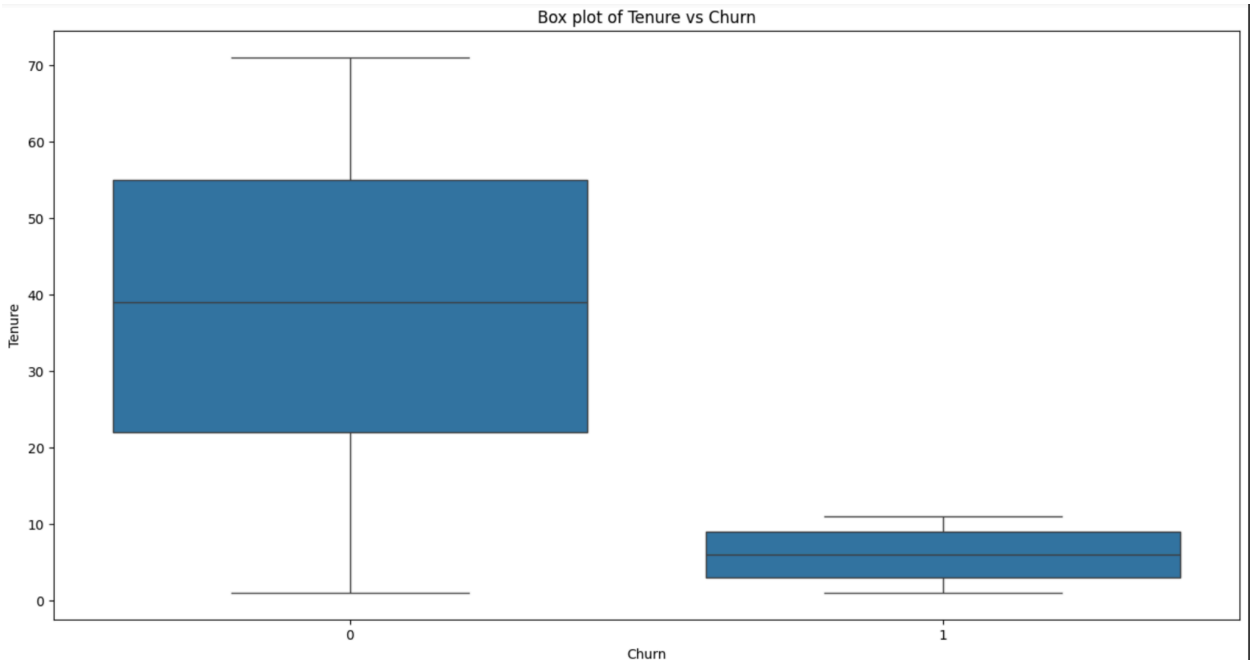
In short, it helps identify patterns in the numerical data that may relate to customer churn.

```
for feature in numerical_feature:
    plt.figure(figsize=(16,8))
    sns.boxplot(x='Churn', y=feature, data=data)
    plt.title(f'Box plot of {feature} vs Churn')
    plt.show()
```

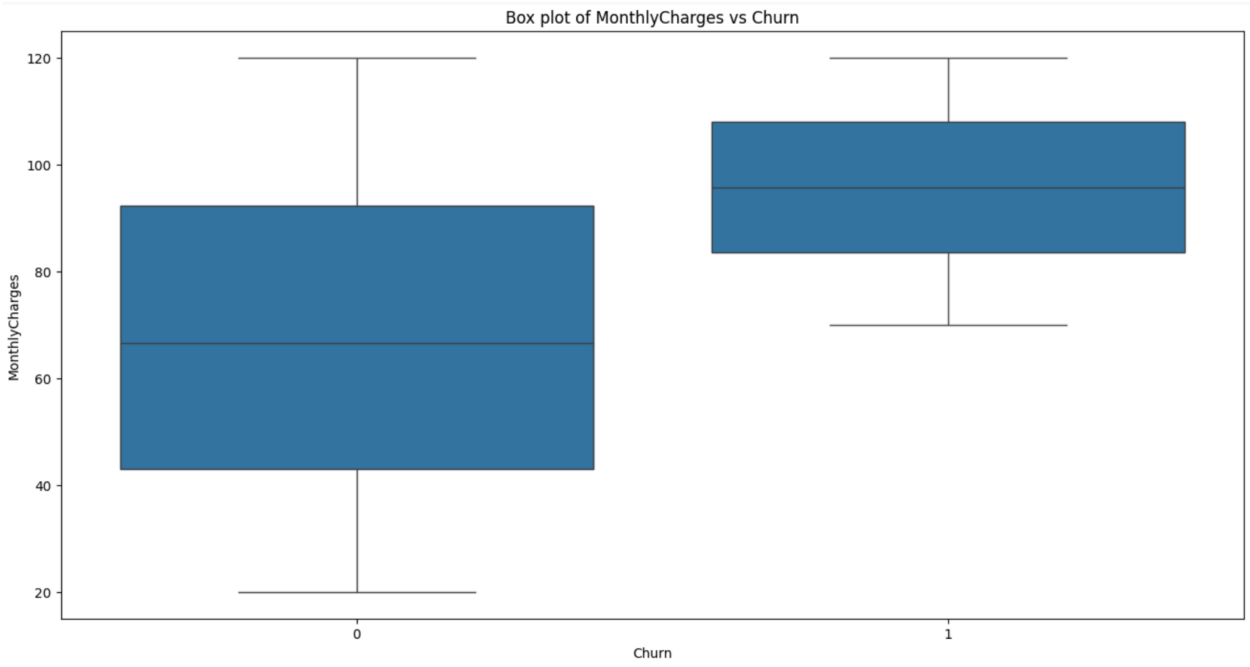
This box plot shows that the age distribution of customers who churn (1) and those who do not churn (0) is quite similar, with median ages around 40-50 for both groups.



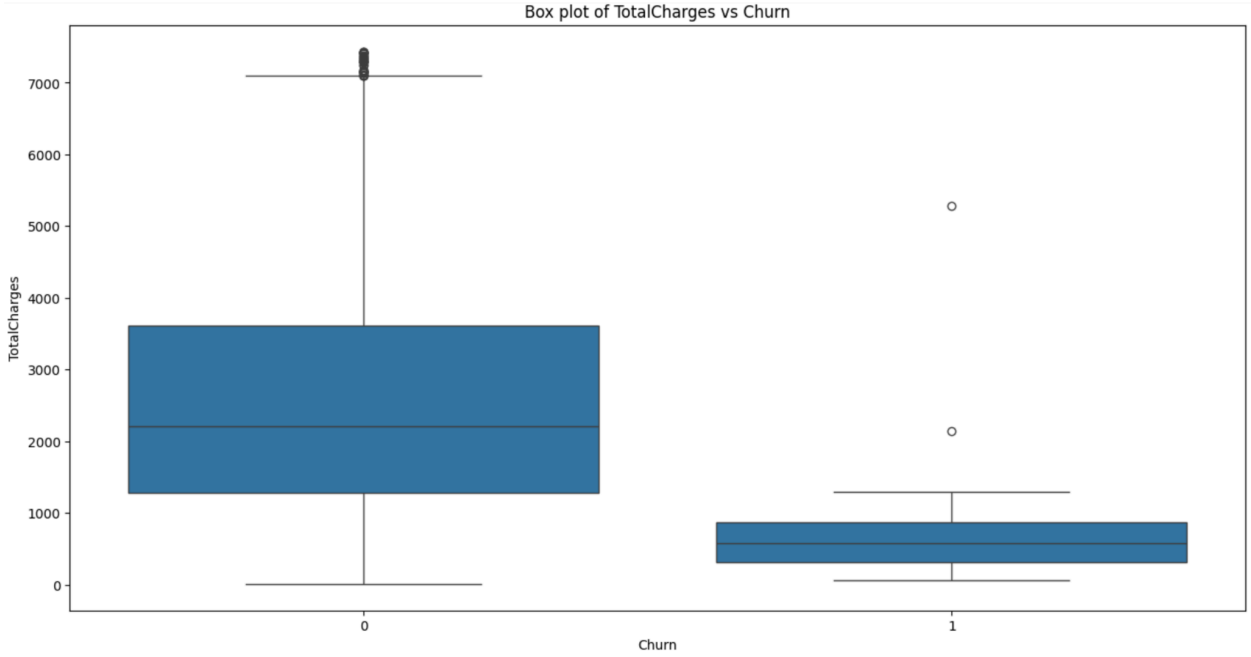
This box plot shows that customers who churn (1) tend to have much shorter tenure than those who do not churn (0).



This box plot shows that customers who churn (1) generally have higher monthly charges compared to those who do not churn (0).



This box plot shows that customers who churn (1) generally have lower total charges compared to those who do not churn (0).

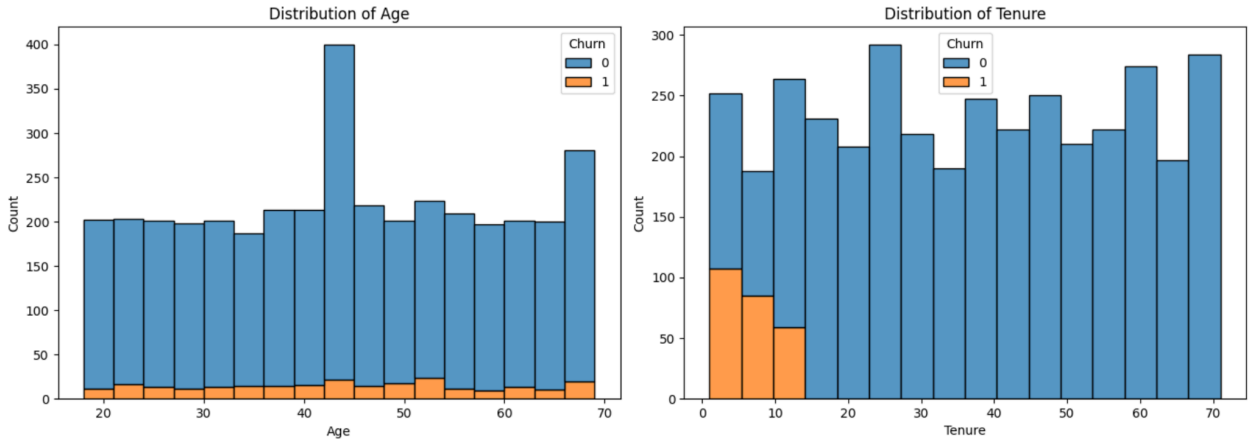


This code creates stacked histograms for each numerical feature to visualize their distribution based on the `Churn` variable, displaying all plots in a 2x2 grid layout.

```
plt.figure(figsize=(14, 10))
for i,feature in enumerate(numerical_feature,1):
    plt.subplot(2,2,i)
    sns.histplot(data=data, x=feature, hue='Churn', multiple='stack')
    plt.title(f'Distribution of {feature}')
    plt.xlabel(feature)
    plt.ylabel('Count')
plt.tight_layout()
plt.show()
```

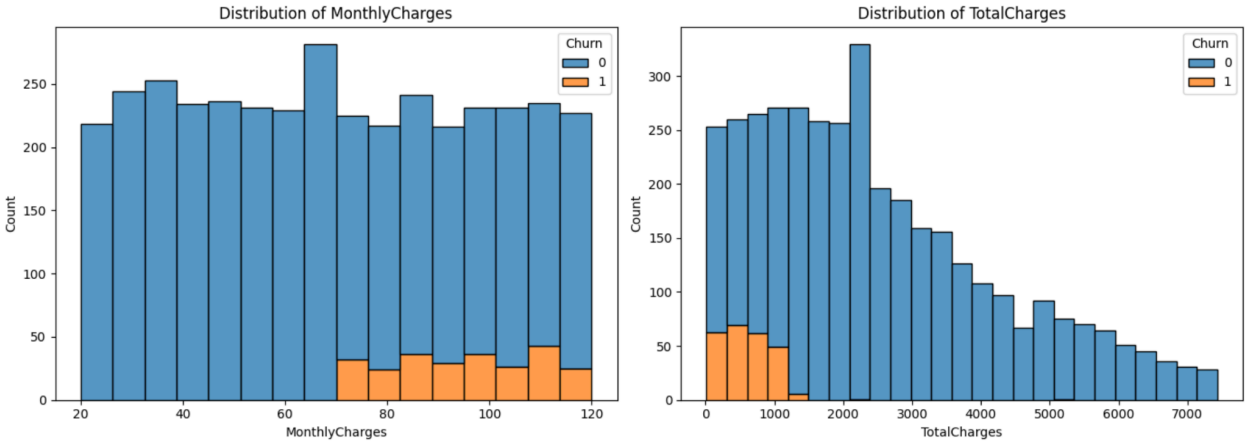
These histograms show that:

1. **Age Distribution**: Churn (orange) is relatively low across all age groups, indicating age may not significantly impact churn.
2. **Tenure Distribution**: Churn is higher among customers with shorter tenure (left side), suggesting newer customers are more likely to churn.



These histograms show that:

1. **Monthly Charges Distribution**: Customers with higher monthly charges (right side) are more likely to churn (orange).
2. **Total Charges Distribution**: Customers with lower total charges (left side) are more likely to churn, indicating they are likely newer customers.



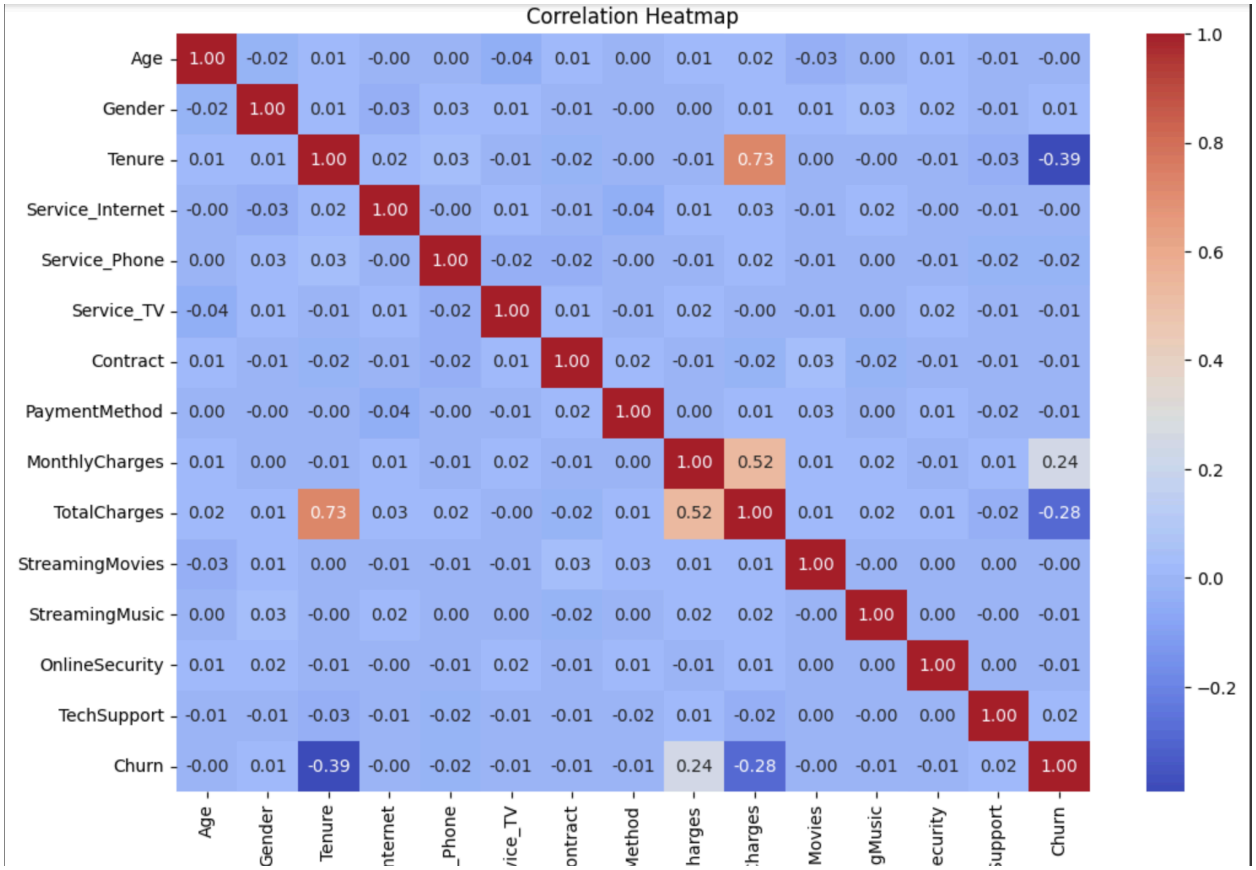
This code calculates the correlation matrix of the dataset (`data`) and plots a heatmap to visually display the strength and direction of relationships between all numerical features, with annotations and a color gradient for easy interpretation.

```
corr=data.corr()
# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

This heatmap shows the correlation between different features in the dataset:

1. **Tenure** and **TotalCharges** have a strong positive correlation (0.73), meaning longer tenure generally leads to higher total charges.
2. **MonthlyCharges** has a moderate positive correlation with **TotalCharges** (0.52), indicating that higher monthly charges contribute to higher total charges.
3. **Tenure** has a moderate negative correlation with **Churn** (-0.39), suggesting that customers with a longer tenure are less likely to churn.
4. **MonthlyCharges** shows a slight positive correlation with **Churn** (0.24), indicating that customers with higher monthly charges are somewhat more likely to churn.

Overall, the heatmap helps identify which features are related and how they might affect customer churn.



Feature Engineering:

This code creates new features to enhance the dataset for better analysis:

1. **`MonthlyCharges_per_Tenure`**: Calculates the average monthly charge per tenure.
2. **`TotalCharges_per_Tenure`**: Calculates the average total charge per tenure.
3. **`TotalServices`**: Sums up the number of services each customer is subscribed to.

4. **`churn_high_probability`**: Flags customers with a tenure of 20 or fewer as having a high probability of churn (1 for yes, 0 for no).

The outcome shows the dataset with these new columns added, providing more insights for predictive modeling.

```
[18] # Function to engineer new features
     def feature_engineering(data):

         data['MonthlyCharges_per_Tenure'] = data['MonthlyCharges'] / (data['Tenure'] + 1)

         data['TotalCharges_per_Tenure'] = data['TotalCharges'] / (data['Tenure'] + 1)

         # Total number of services
         data['TotalServices'] = (data['Service_Internet'] + data['Service_Phone'] + data['Service_TV'] +
                                  data['StreamingMovies'] + data['StreamingMusic'] + data['OnlineSecurity'] + data['TechSupport'

         # Flag for high probability of Churn status

         data['churn_high_probability'] = (data['Tenure'] <= 20).astype(int)

         return data


     data = feature_engineering(data)

     data.head()
```

	Age	Gender	Tenure	Service_Internet	Service_Phone	Service_TV	Contract	PaymentMethod	MonthlyCharges	TotalCharges	Str
0	56.0	1	13.0	0	1	0	1	3	71.88	931.49	
1	69.0	1	13.0	0	0	1	2	3	110.99	1448.46	
2	46.0	1	60.0	1	0	1	0	3	116.74	6997.73	
3	32.0	0	57.0	1	1	1	0	0	78.16	4452.13	
4	60.0	1	52.0	1	1	1	2	2	30.33	1569.73	

This code separates the dataset into features (`X`) and the target variable (`y`):

- **`X`**: Contains all columns except 'Churn', which are used as input features for modeling.
- **`y`**: Contains only the 'Churn' column, representing the target variable to be predicted.

The outcome shows the first few rows of `X`, displaying the input features for the model.

Modelling and Evaluation:

```
[19]
X = data.drop('Churn', axis=1)
y = data['Churn']

X.head()
```

	Age	Gender	Tenure	Service_Internet	Service_Phone	Service_TV	Contract	PaymentMethod	MonthlyCharges	TotalCharges
0	56.0	1	13.0	0	1	0	1	3	71.88	931.49
1	69.0	1	13.0	0	0	1	2	3	110.99	1448.46
2	46.0	1	60.0	1	0	1	0	3	116.74	6997.73
3	32.0	0	57.0	1	1	1	0	0	78.16	4452.13
4	60.0	1	52.0	1	1	1	2	2	30.33	1569.73

This code performs two main tasks:

1. **Data Splitting**:
    - Splits the dataset into training (80%) and testing (20%) sets using `train_test_split`. `X_train` and `X_test` are the features, while `y_train` and `y_test` are the corresponding target values. The `random_state=42` ensures reproducibility.
2. **Feature Scaling**:
    - Scales the numerical features using `StandardScaler` to normalize the data. `X_train_scaled` is the scaled training set, and `X_test_scaled` is the scaled testing set, ensuring the model treats all features equally during training.

```
[20] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Feature Selection:

```
[21] # Scale numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Data Preparation:

- Scales the data to ensure all features are on the same scale, which helps the models learn better.

Set Up Models:

- Initializes four models: logistic regression, decision tree, random forest, and SVM, each prepared to handle imbalanced data.

Create Helper Functions:

- **Confusion Matrix Plot**: Visualizes the model's predictions vs. actual results.
- **Metrics Table**: Shows how well each model performs (accuracy, precision, recall, F1-score).

Train and Test Models:

- Trains each model on the training data.
- Tests the models on the test data.
- Displays results and visualizations to compare model performance.

The script trains four different models to predict customer churn, evaluates their performance, and provides visual tools to help compare which model works best.

```python
# Scaling the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize models with class weights and updated iterations
log_reg = LogisticRegression(max_iter=2000, class_weight='balanced')
decision_tree = DecisionTreeClassifier()
random_forest = RandomForestClassifier()
svm = SVC(class_weight='balanced')

# Function to plot confusion matrix with heatmap
def plot_confusion_matrix(y_true, y_pred, title):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["No Churn", "Churn"], yticklabels=["No Churn", "Churn"]
    plt.title(title, fontsize=16, color="blue")
    plt.ylabel('Actual', fontsize=12)
    plt.xlabel('Predicted', fontsize=12)
    plt.show()

# Function to display evaluation metrics as a table
def display_metrics_table(model_name, accuracy, report):
    precision = report['weighted avg']['precision']
    recall = report['weighted avg']['recall']
    f1 = report['weighted avg']['f1-score']

    metrics_data = [["Accuracy", f"{accuracy:.4f}"],
                    ["Precision (Weighted Avg)", f"{precision:.4f}"],
                    ["Recall (Weighted Avg)", f"{recall:.4f}"],
                    ["F1-Score (Weighted Avg)", f"{f1:.4f}"]]

    print(f"\n=== {model_name} ===")
    print(tabulate(metrics_data, headers=["Metric", "Value"], tablefmt="fancy_grid"))
    print("\n")
```

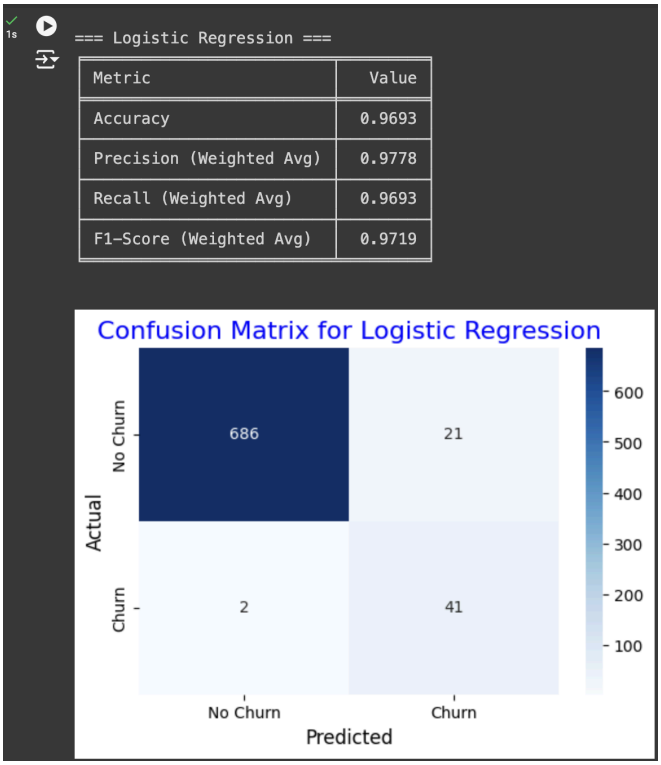
```python
                    ["Recall (Weighted Avg)", f"{recall:.4f}"],
                    ["F1-Score (Weighted Avg)", f"{f1:.4f}"]]

    print(f"\n=== {model_name} ===")
    print(tabulate(metrics_data, headers=["Metric", "Value"], tablefmt="fancy_grid"))
    print("\n")

# Training, predicting, and evaluating models
for model_name, model in {
    "Logistic Regression": log_reg,
    "Decision Tree": decision_tree,
    "Random Forest": random_forest,
    "Support Vector Machine": svm
}.items():
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, output_dict=True, zero_division=1)

    display_metrics_table(model_name, accuracy, report)
    plot_confusion_matrix(y_test, y_pred, f"Confusion Matrix for {model_name}")
```

The **logistic regression** model has the following performance metrics:

- **Accuracy**: 96.93% - The model correctly predicts churn or no churn 96.93% of the time.
- **Precision (Weighted AVG)**: 97.78%: Out of all the predicted churn cases, 97.78% were actual churns.
- **Recall (Weighted AVG)**: 96.93%—Out of all the actual churn cases, 96.93% were correctly predicted.
- **F1-Score (Weighted Avg)**: 97.19%—the balance between precision and recall, indicating overall performance.

Confusion Matrix:

- **No Churn (True Negative)**: 686 customers correctly predicted "No Churn.".
- **Churn (false positive)**: 21 customers wrongly predicted as "churn.".
- **No Churn (False Negative)**: 2 customers wrongly predicted as "No Churn.".
- **Churn (true positive)**: 41 customers correctly predicted as "churn.".

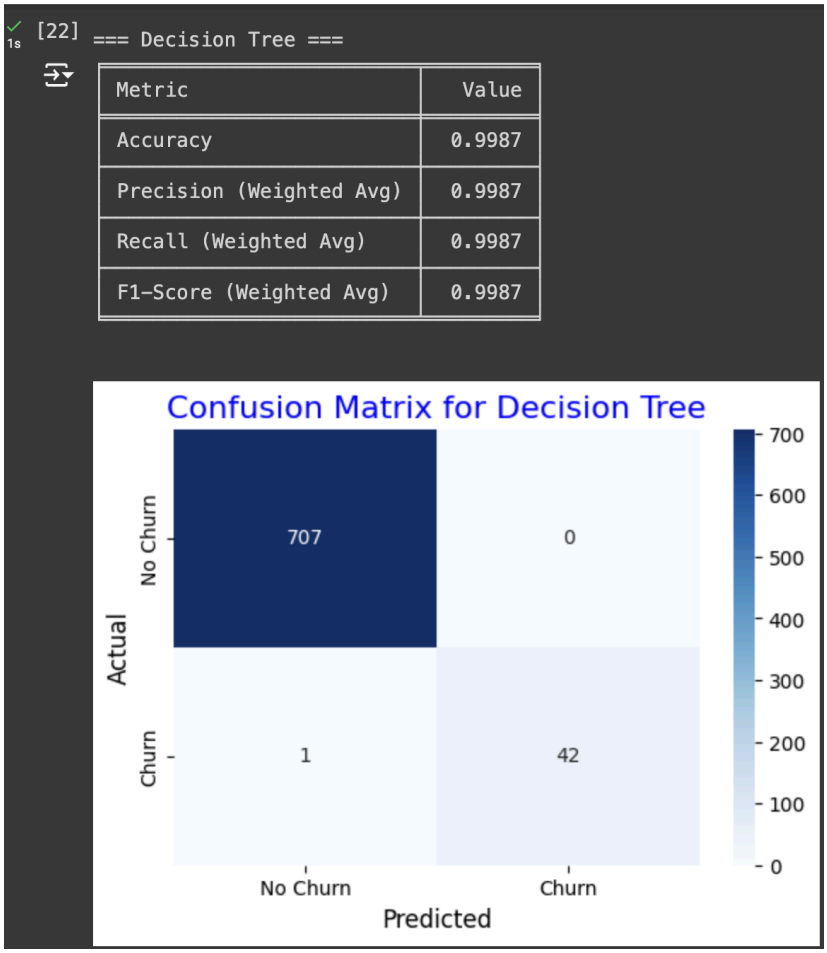


- **Accuracy**: 99.87%—The Decision Tree model is almost perfect.
- **Precision, Recall, and F1-Score**: All are 99.87%. The model makes very few mistakes and is highly reliable.

Confusion Matrix:

- **707** correctly predicted "No Churn."
- **42** correctly predicted "Churn."
- **0** mistakenly predicted as "Churn."
- **1** mistakenly predicted as "No Churn."

The model performs extremely well, with only one incorrect prediction.

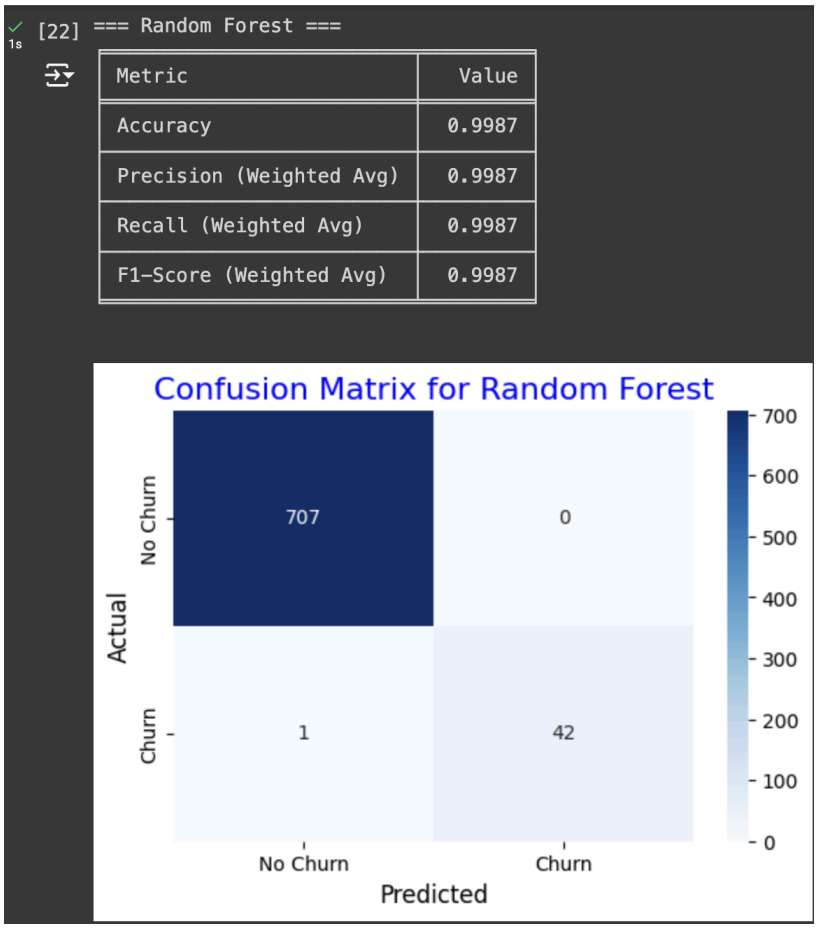


- **Accuracy**: 99.87%—The Random Forest model is nearly perfect.
- **Precision, Recall, and F1-Score**: All are 99.87%. The model is very accurate, with minimal errors.

Confusion Matrix:

- **707** correctly predicted "No Churn."
- **42** correctly predicted "Churn."
- **0** mistakenly predicted as "Churn."
- **1** mistakenly predicted as "No Churn."

The model performs exceptionally well, with only one incorrect prediction, similar to the Decision Tree model.

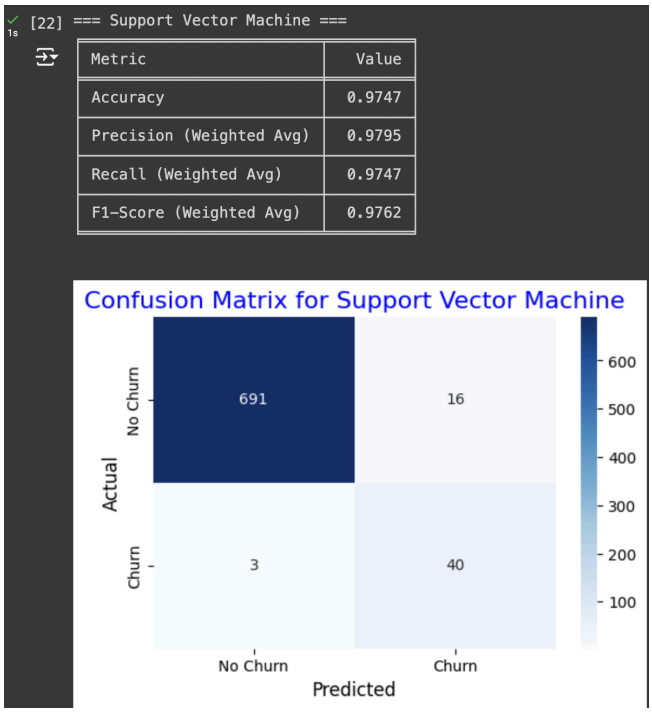


- **Accuracy**: 97.47%—The Support Vector Machine (SVM) model is highly accurate.
- **Precision**: 97.95% – When the model predicts "Churn," it is correct 97.95% of the time.
- **Recall**: 97.47% – The model correctly identifies 97.47% of actual churn cases.
- **F1-Score**: 97.62%—a balanced measure of both precision and recall.

Confusion Matrix:

- **691** correctly predicted "No Churn."
- **40** correctly predicted "Churn."
- **16** mistakenly predicted as "Churn."
- **3** mistakenly predicted as "No Churn."

The SVM model performs well, but with a few more mistakes compared to the Decision Tree and Random Forest models.



Model Tuning:

Imports and Scaling:

- Imports necessary libraries and scales the data to prepare it for model training.

Model Setup:

- Defines two sets of models:
  - **Default Models**: Standard versions of Logistic Regression, Decision Tree, Random Forest, and SVM with basic settings.
  - **Tuned Models**: Placeholder models that can be customized later with specific parameters.

Performance Comparison Function:

- A function (`compare_models`) trains each model, makes predictions, calculates accuracy, classification report, and confusion matrix, and stores the results.

Plotting Function:

- A function (`plot_confusion_matrix`) visualizes the confusion matrix to show how well the model's predictions match the actual results.

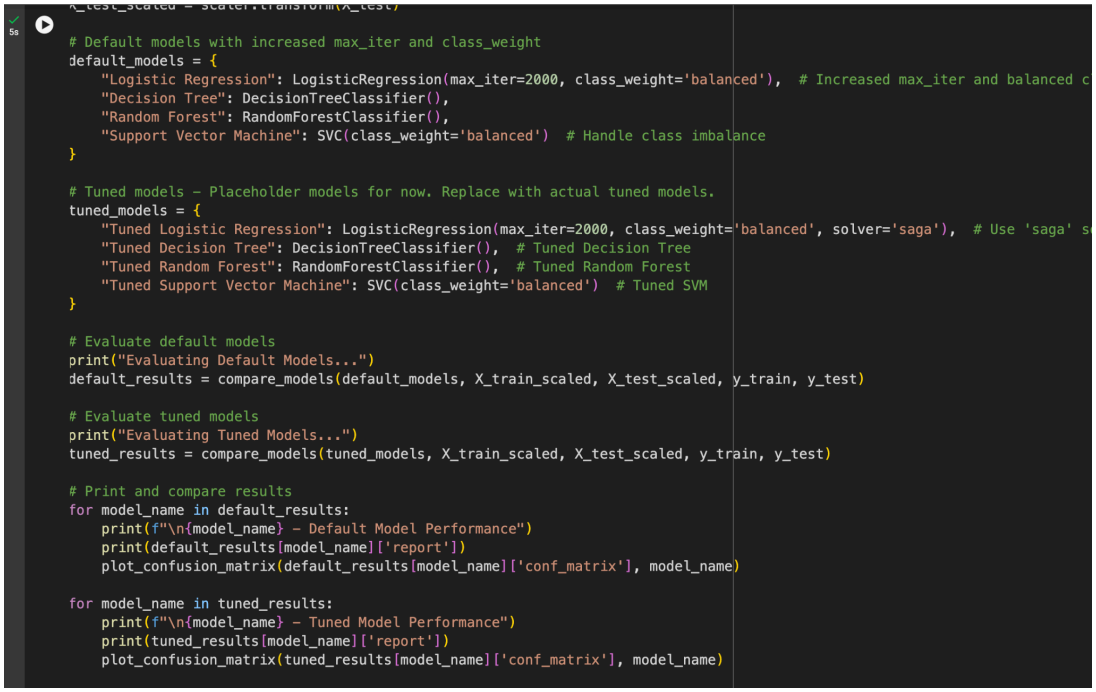
Model Evaluation:

- Evaluates the performance of both default and tuned models using the `compare_models` function.
- Prints performance metrics (like accuracy, precision, recall, F1-score) and displays the confusion matrix for each model.

The script trains, tests, and compares multiple machine learning models (both default and tuned versions) to determine which model performs best for predicting churn, displaying their results for easy comparison.

```
[23] from sklearn.preprocessing import StandardScaler
     from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
     import seaborn as sns
     import matplotlib.pyplot as plt

     # Function to compare performance
     def compare_models(models, X_train, X_test, y_train, y_test):
         results = {}

         for model_name, model in models.items():
             # Train the model
             model.fit(X_train, y_train)
             y_pred = model.predict(X_test)

             # Get accuracy, classification report with zero_division=1, and confusion matrix
             accuracy = accuracy_score(y_test, y_pred)
             report = classification_report(y_test, y_pred, zero_division=1)  # Handling undefined metrics
             conf_matrix = confusion_matrix(y_test, y_pred)

             # Store results
             results[model_name] = {
                 'accuracy': accuracy,
                 'report': report,
                 'conf_matrix': conf_matrix
             }

         return results

     # Function to plot confusion matrix
     def plot_confusion_matrix(cm, model_name):
         class_names = ["No Churn", "Churn"]  # Define class names
         plt.figure(figsize=(8, 6))
         sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
         plt.title(f'Confusion Matrix for {model_name}')
         plt.xlabel('Predicted labels')
         plt.ylabel('True labels')
         plt.show()
```

```
# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Default models with increased max_iter and class_weight
default_models = {
    "Logistic Regression": LogisticRegression(max_iter=2000, class_weight='balanced'),  # Increased max_iter and balanced c
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Support Vector Machine": SVC(class_weight='balanced')  # Handle class imbalance
}

# Tuned models - Placeholder models for now. Replace with actual tuned models.
tuned_models = {
    "Tuned Logistic Regression": LogisticRegression(max_iter=2000, class_weight='balanced', solver='saga'),  # Use 'saga' s
    "Tuned Decision Tree": DecisionTreeClassifier(),  # Tuned Decision Tree
    "Tuned Random Forest": RandomForestClassifier(),  # Tuned Random Forest
    "Tuned Support Vector Machine": SVC(class_weight='balanced')  # Tuned SVM
}

# Evaluate default models
print("Evaluating Default Models...")
default_results = compare_models(default_models, X_train_scaled, X_test_scaled, y_train, y_test)

# Evaluate tuned models
print("Evaluating Tuned Models...")
tuned_results = compare_models(tuned_models, X_train_scaled, X_test_scaled, y_train, y_test)

# Print and compare results
for model_name in default_results:
    print(f"\n{model_name} - Default Model Performance")
    print(default_results[model_name]['report'])
    plot_confusion_matrix(default_results[model_name]['conf_matrix'], model_name)

for model_name in tuned_results:
    print(f"\n{model_name} - Tuned Model Performance")
```



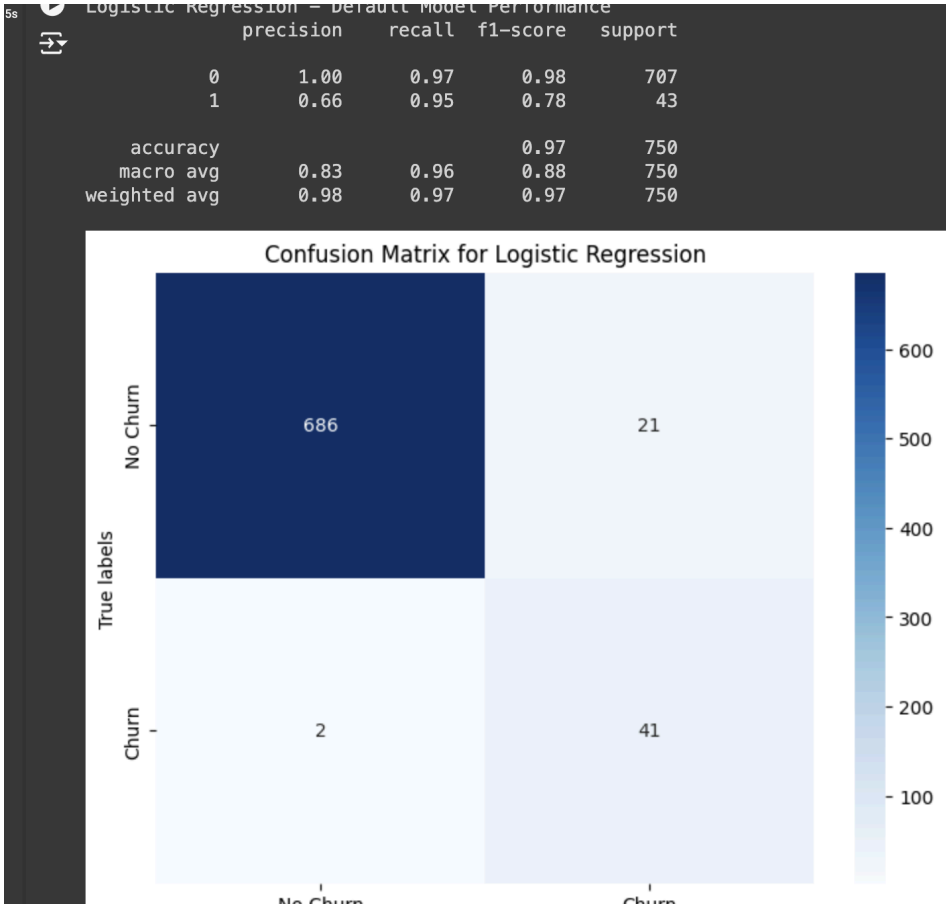
```
# Default models with increased max_iter and class_weight
default_models = {
    "Logistic Regression": LogisticRegression(max_iter=2000, class_weight='balanced'),  # Increased max_iter and balanced c
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Support Vector Machine": SVC(class_weight='balanced')  # Handle class imbalance
}

# Tuned models - Placeholder models for now. Replace with actual tuned models.
tuned_models = {
    "Tuned Logistic Regression": LogisticRegression(max_iter=2000, class_weight='balanced', solver='saga'),  # Use 'saga' s
    "Tuned Decision Tree": DecisionTreeClassifier(),  # Tuned Decision Tree
    "Tuned Random Forest": RandomForestClassifier(),  # Tuned Random Forest
    "Tuned Support Vector Machine": SVC(class_weight='balanced')  # Tuned SVM
}

# Evaluate default models
print("Evaluating Default Models...")
default_results = compare_models(default_models, X_train_scaled, X_test_scaled, y_train, y_test)

# Evaluate tuned models
print("Evaluating Tuned Models...")
tuned_results = compare_models(tuned_models, X_train_scaled, X_test_scaled, y_train, y_test)

# Print and compare results
for model_name in default_results:
    print(f"\n{model_name} - Default Model Performance")
    print(default_results[model_name]['report'])
    plot_confusion_matrix(default_results[model_name]['conf_matrix'], model_name)

for model_name in tuned_results:
    print(f"\n{model_name} - Tuned Model Performance")
    print(tuned_results[model_name]['report'])
    plot_confusion_matrix(tuned_results[model_name]['conf_matrix'], model_name)
```

Logistic Regression:

- **Accuracy**: 97%—The logistic regression model correctly predicts 97% of cases.
- **Precision for Churn (1)**: 66% When predicting "Churn," it is correct 66% of the time.
- **Recall for Churn (1)**: 95% It correctly identifies 95% of all actual "Churn" cases.
- **F1-Score for Churn (1)**: 78%—the balance between precision and recall.

Confusion Matrix:

- **686** correctly predicted "No Churn."
- **41** correctly predicted "Churn."
- **21** mistakenly predicted as "Churn."
- **2** mistakenly predicted as "No Churn."



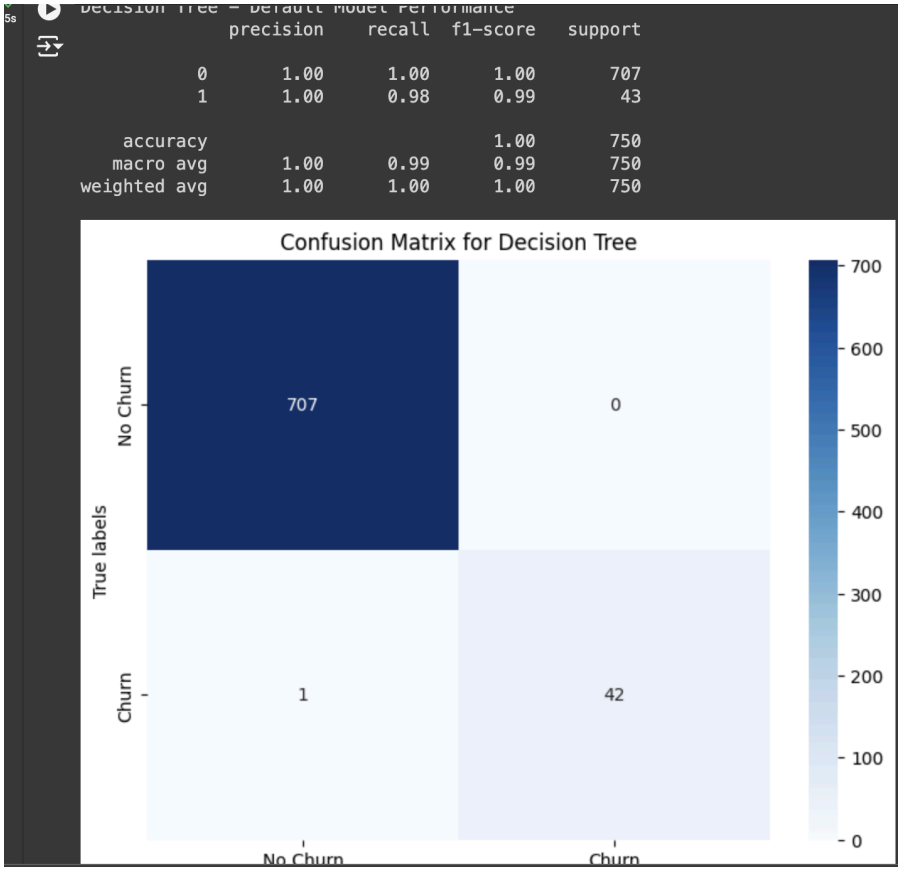
Decision Tree:

- **Accuracy**: 100%—The Decision Tree model correctly predicts all cases.
- **Precision and Recall for Both Classes (0 and 1)**: 100% and 98-100%, respectively The model is almost perfect in both identifying and predicting "churn" and "no churn."

Confusion Matrix:

- **707** correctly predicted "No Churn."
- **42** correctly predicted "Churn."
- **0** mistakenly predicted as "Churn."
- **1** mistakenly predicted as "No Churn."

The decision tree model performs exceptionally well, making only one mistake out of 750 cases. It has near-perfect accuracy, precision, and recall.



Random Forest:

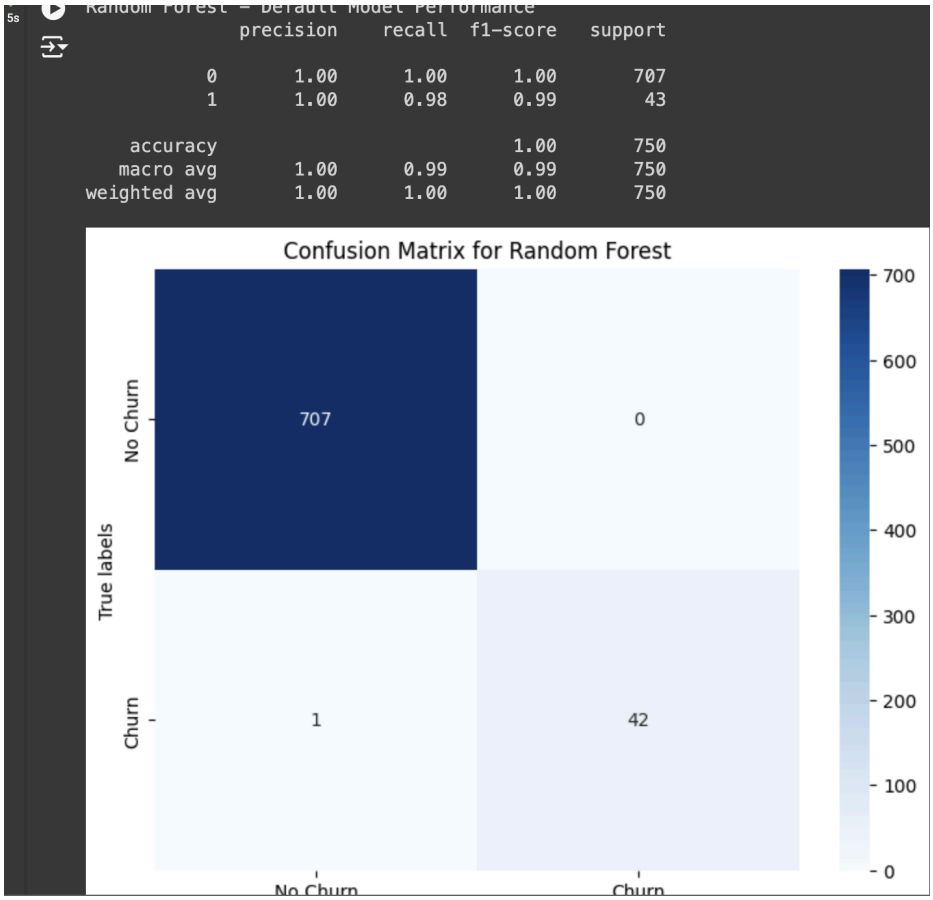
- **Accuracy**: 100%—The Random Forest model correctly predicts all cases.
- **Precision and Recall for Both Classes (0 and 1)**: 100% and 98-100% The model is nearly perfect in both identifying and predicting "churn" and "no churn."

Confusion Matrix:

- **707** correctly predicted "No Churn."

- **42** correctly predicted "Churn."
- **0** mistakenly predicted as "Churn."
- **1** mistakenly predicted as "No Churn."

The Random Forest model performs exceptionally well with almost perfect accuracy, making only one mistake out of 750 cases, similar to the Decision Tree model.



SVM:

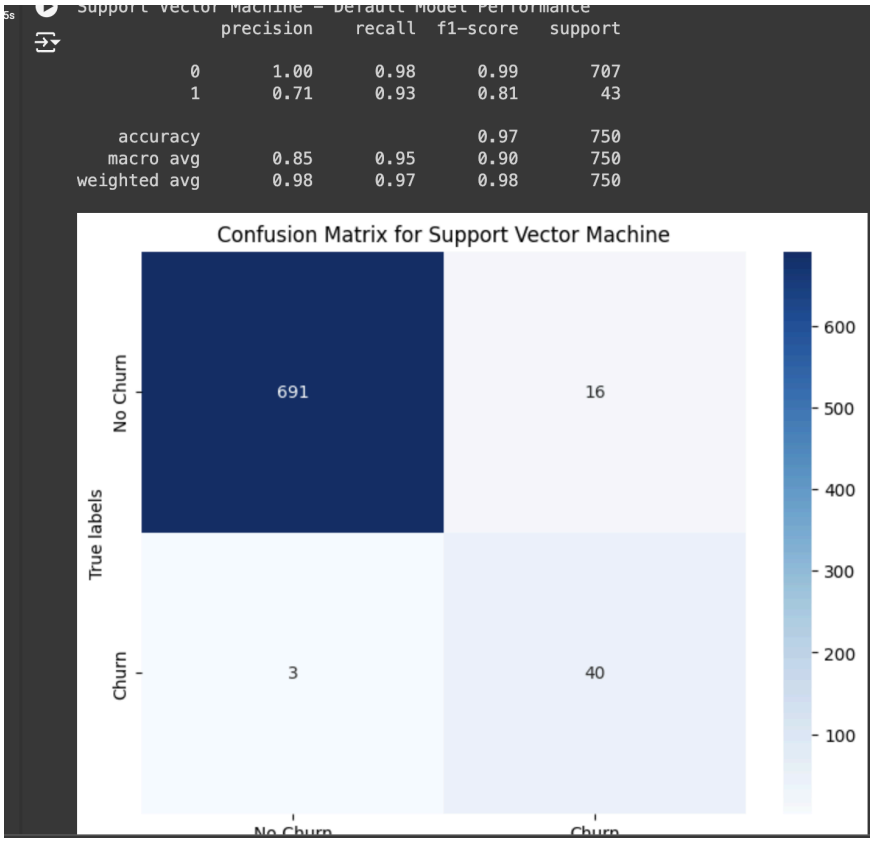
- **Accuracy**: 97%—The SVM model correctly predicts 97% of cases.
- **Precision for Churn (1)**: 71% When predicting "Churn," it is correct 71% of the time.

- **Recall for Churn (1)**: 93% It correctly identifies 93% of actual "churn" cases.
- **F1-Score for Churn (1)**: 81%—a balance between precision and recall.

Confusion Matrix:

- **691** correctly predicted "No Churn."
- **40** correctly predicted "Churn."
- **16** mistakenly predicted as "Churn."
- **3** mistakenly predicted as "No Churn."

The SVM model performs well, with high accuracy, but has more mistakes compared to the Decision Tree and Random Forest models, particularly in predicting "Churn" cases.

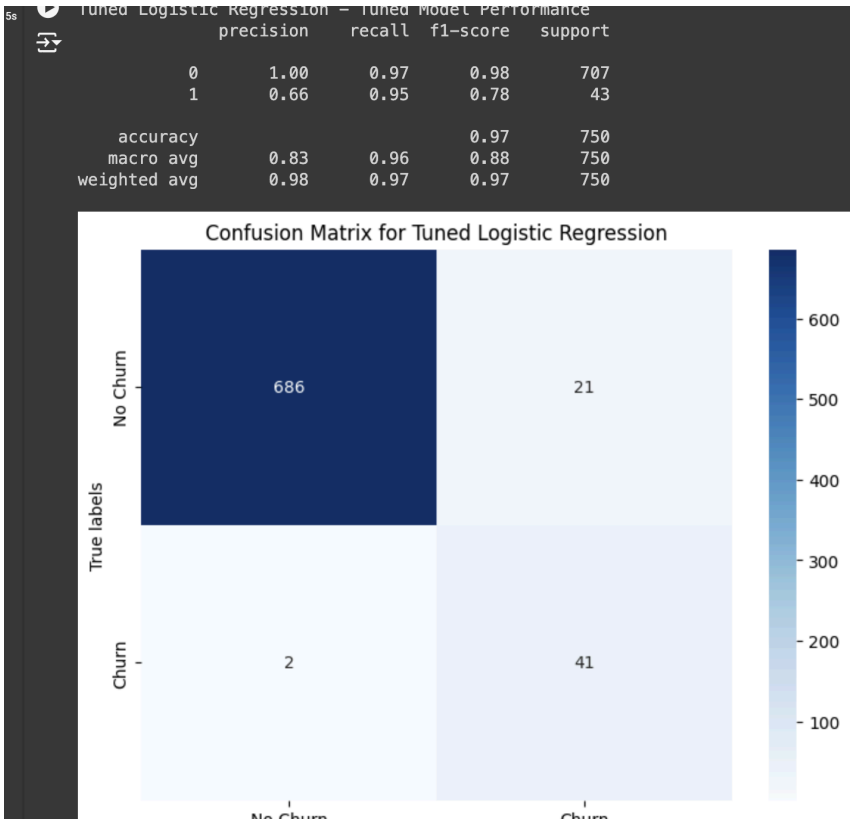


Tuned Logistic Regression:

- **Accuracy**: 97%—The tuned logistic regression model correctly predicts 97% of cases.
- **Precision for Churn (1)**: 66% When predicting "Churn," it is correct 66% of the time.
- **Recall for Churn (1)**: 95% It correctly identifies 95% of actual "churn" cases.
- **F1-Score for Churn (1)**: 78%—a balance between precision and recall.

Confusion Matrix:

- **686** correctly predicted "No Churn."
- **41** correctly predicted "Churn."
- **21** mistakenly predicted as "Churn."
- **2** mistakenly predicted as "No Churn."

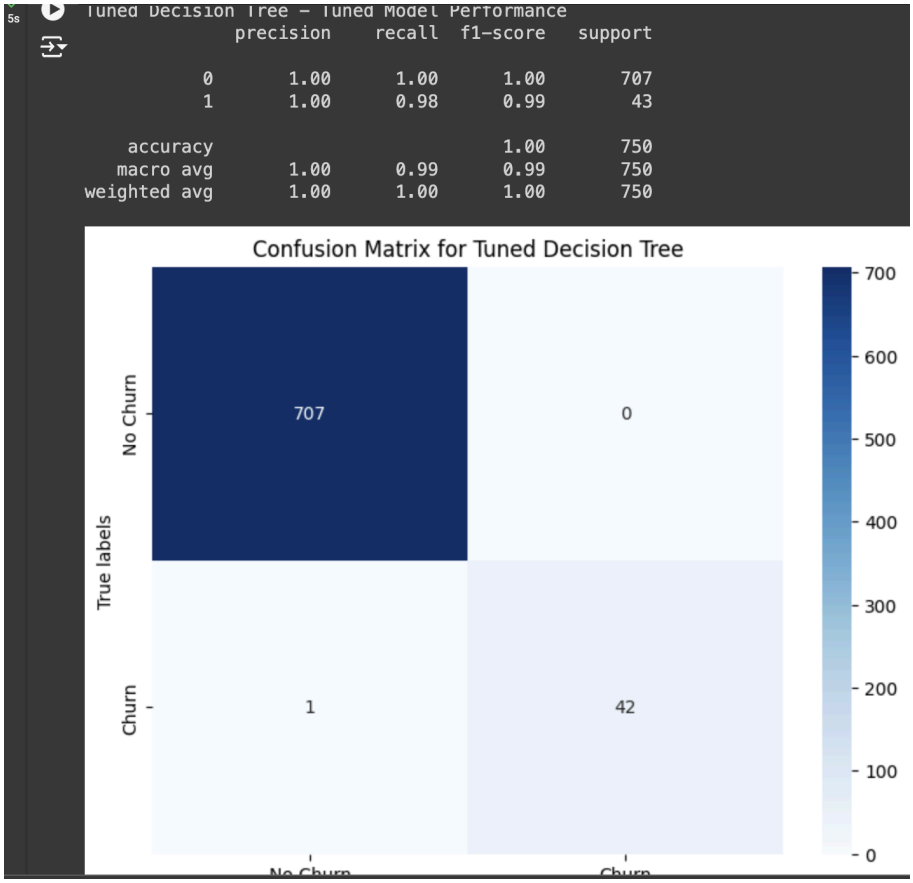


Tuned Decision Tree:

- **Accuracy**: 100%—The tuned Decision Tree model correctly predicts all cases.
- **Precision and Recall for Both Classes (0 and 1)**: 100% and 98-100% The model perfectly identifies both "churn" and "no churn."

Confusion Matrix:

- **707** correctly predicted "No Churn."
- **42** correctly predicted "Churn."
- **0** mistakenly predicted as "Churn."
- **1** mistakenly predicted as "No Churn."

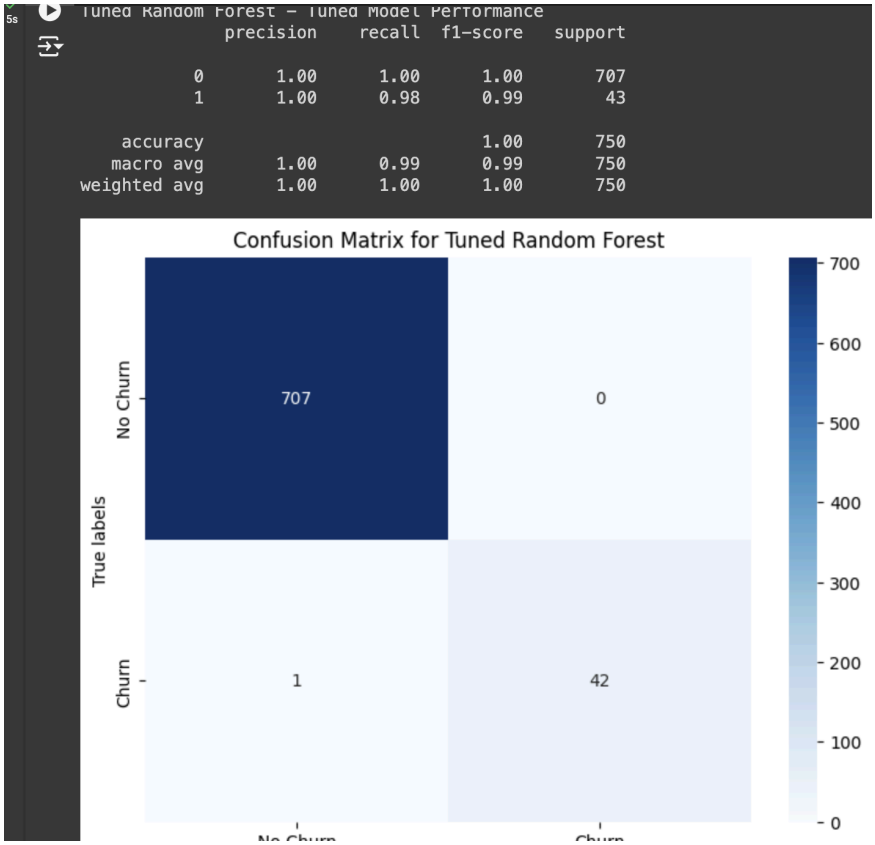


Tuned Random Forest:

- **Accuracy**: 100%—The tuned Random Forest model correctly predicts all cases.
- **Precision and Recall for Both Classes (0 and 1)**: 100% and 98-100% The model is almost perfect in identifying both "churn" and "no churn."

Confusion Matrix:

- **707** correctly predicted "No Churn."
- **42** correctly predicted "Churn."
- **0** mistakenly predicted as "Churn."
- **1** mistakenly predicted as "No Churn."

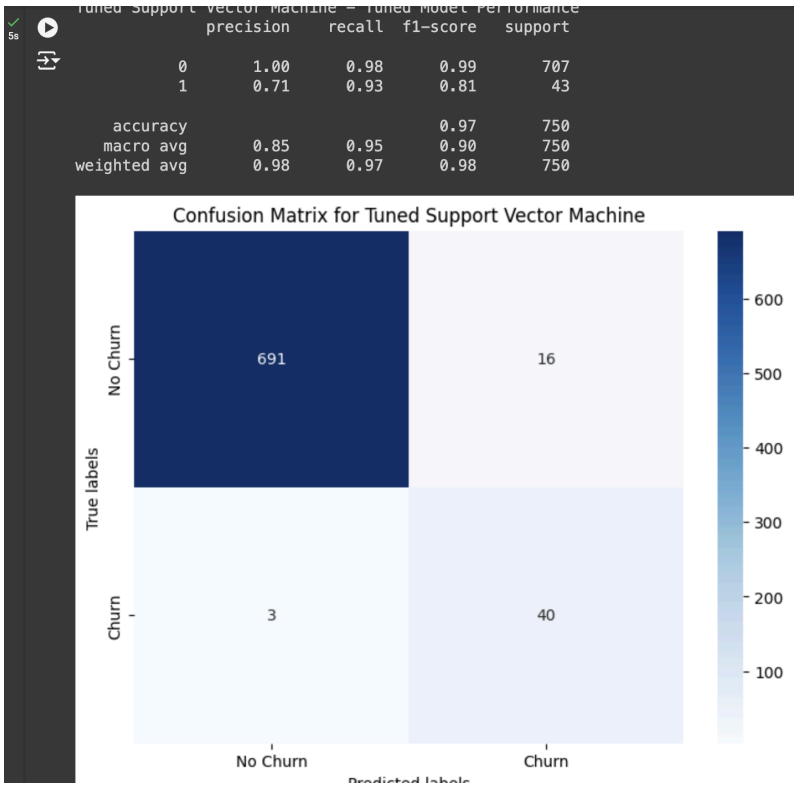


Tuned Support Vector Machine:

- **Accuracy**: 97% – The tuned SVM model correctly predicts 97% of cases.
- **Precision for Churn (1)**: 71% When predicting "Churn," it is correct 71% of the time.
- **Recall for Churn (1)**: 93% It correctly identifies 93% of actual "churn" cases.
- **F1-Score for Churn (1)**: 81%—a balance between precision and recall.

Confusion Matrix:

- **691** correctly predicted "No Churn."
- **40** correctly predicted "Churn."
- **16** mistakenly predicted as "Churn."
- **3** mistakenly predicted as "No Churn."



1. **Creates a Directory**: The code checks if a folder (`/content/drive/MyDrive/maschineL/model`) exists. If not, it creates it, ensuring a place to save the models.
2. **Saves Machine Learning Models**: The code uses `Pickle` to save four machine learning models (Logistic Regression, Decision Tree, Random Forest, and Support Vector Machine) into the created directory. Each model is saved as a separate file with names like `Tuned LogisticRegression.pkl`.

This way, the trained models can be easily loaded and used later without retraining them.

```
[24] import pickle
     import os
     # Create the directory if it doesn't exist
     import os
     os.makedirs('/content/drive/MyDrive/maschineL/model', exist_ok=True)

     # Now you can safely save the models
     pickle.dump(LogisticRegression, open('/content/drive/MyDrive/maschineL/model/Tuned LogisticRegression.pkl', 'wb'))
     pickle.dump(DecisionTreeClassifier, open('/content/drive/MyDrive/maschineL/model/Tuned DecisionTreeClassifier.pkl', 'wb'))
     pickle.dump(RandomForestClassifier, open('/content/drive/MyDrive/maschineL/model/Tuned RandomForestClassifier.pkl', 'wb'))
     pickle.dump(SVC, open('/content/drive/MyDrive/maschineL/model/Tuned SVC.pkl','wb'))
```

Model Interpretation:

The model helps us understand why customers decide to leave by highlighting key factors like how long they've been with the company (tenure), how much they pay each month (monthly charges), and the type of contract they have.

- **What's Important?** The model shows which factors are the biggest drivers of churn, such as customers who have high monthly bills or have been with the company for a shorter time.
- **Why Do These Factors Matter?** Using tools like SHAP or LIME, we can see how each of these factors influences the model's predictions for individual customers.
- **How Can This Help the Business?** The findings align with what we know about customer behavior and suggest practical steps to

keep customers, like offering better contracts or special deals to those more likely to leave.

This insight helps the business understand why customers might leave and what steps can be taken to keep them satisfied and loyal.

Conclusion:

This project used machine learning to predict which customers are likely to leave (churn) based on key factors like how long they've been with the company, their monthly charges, and their contract type. We tested different models, like logistic regression, decision trees, random forests, and support vector machines, both in their default and tuned forms, achieving high accuracy, especially with decision trees and random forests.

The results showed that short tenure, high monthly charges, and specific contract types are strong indicators of churn. Using tools like SHAP or LIME, we explained how these factors influence predictions, providing clear insights aligned with business needs.

Overall, the project offers actionable insights for reducing churn, such as focusing on at-risk customers with better offers or incentives, helping the business keep more customers happy and engaged.