Name = Manoj Kumar

Batch = 1st September Batch

Course = Data Science Placement Guarantee Course

Email = manojkumarrajput@gmail.com

Part 2 Explaination Video link = https://drive.google.com/file/d/1vzkcS44vJWz_A-AxplpyQqSZYUGzthJH/view?usp=drive_link

Importing the required packages

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
!pip install xgboost
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

```
Requirement already satisfied: xgboost in c:\users\msgme\anaconda3\lib\site-packages
(2.1.3)
Requirement already satisfied: numpy in c:\users\msgme\anaconda3\lib\site-packages
(from xgboost) (1.26.4)
Requirement already satisfied: scipy in c:\users\msgme\anaconda3\lib\site-packages
(from xgboost) (1.13.1)
```

Reading and Exploring Airbnb Dataset

EDA

Data Inspection

In [3]:
```
Cx_DataSet = pd.read_csv("C:\\Users\\msgme\\Downloads\\Customer_data.csv")
```

Displayed the top 5 rows of the data, so that we can get a quick view of the data.

In [5]:
```
Cx_DataSet.head()
```

Out[5]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	Multipl
0	7590-VHVEG	Female	0	Yes	No	1	No	No
1	5575-GNVDE	Male	0	No	No	34	Yes	
2	3668-QPYBK	Male	0	No	No	2	Yes	
3	7795-CFOCW	Male	0	No	No	45	No	No
4	9237-HQITU	Female	0	No	No	2	Yes	

5 rows × 21 columns

Displayed the bottom 5 rows of the data, so that we can get a quick view of the some bottom raws of the data.

In [7]:
```
Cx_DataSet.tail()
```

Out[7]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	Mul
7038	6840-RESVB	Male	0	Yes	Yes	24	Yes	
7039	2234-XADUH	Female	0	Yes	Yes	72	Yes	
7040	4801-JZAZL	Female	0	Yes	Yes	11	No	
7041	8361-LTMKD	Male	1	Yes	No	4	Yes	
7042	3186-AJIEK	Male	0	No	No	66	Yes	

5 rows × 21 columns

In [9]:
```
## here we can see some basic information about data such as data type of the colum
```

In [11]:
```
Cx_DataSet.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customerID        7043 non-null   object 
 1   gender            7043 non-null   object 
 2   SeniorCitizen     7043 non-null   int64  
 3   Partner           7043 non-null   object 
 4   Dependents        7043 non-null   object 
 5   tenure            7043 non-null   int64  
 6   PhoneService      7043 non-null   object 
 7   MultipleLines     7043 non-null   object 
 8   InternetService   7043 non-null   object 
 9   OnlineSecurity    7043 non-null   object 
 10  OnlineBackup      7043 non-null   object 
 11  DeviceProtection  7043 non-null   object 
 12  TechSupport       7043 non-null   object 
 13  StreamingTV       7043 non-null   object 
 14  StreamingMovies   7043 non-null   object 
 15  Contract          7043 non-null   object 
 16  PaperlessBilling  7043 non-null   object 
 17  PaymentMethod     7043 non-null   object 
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7032 non-null   float64
 20  Churn             7043 non-null   object 
dtypes: float64(2), int64(2), object(17)
memory usage: 1.1+ MB
```

In [13]:
```
Cx_DataSet.describe()
```

Out[13]:

	SeniorCitizen	tenure	MonthlyCharges	TotalCharges
count	7043.000000	7043.000000	7043.000000	7032.000000
mean	0.162147	32.371149	64.761692	2283.300441
std	0.368612	24.559481	30.090047	2266.771362
min	0.000000	0.000000	18.250000	18.800000
25%	0.000000	9.000000	35.500000	401.450000
50%	0.000000	29.000000	70.350000	1397.475000
75%	0.000000	55.000000	89.850000	3794.737500
max	1.000000	72.000000	118.750000	8684.800000

Checking if there are any duplicate rows in the dataset.

In [15]:
```
Cx_DataSet.duplicated().sum()
```

Out[15]:
```
0
```

There were no duplicates in the customer data

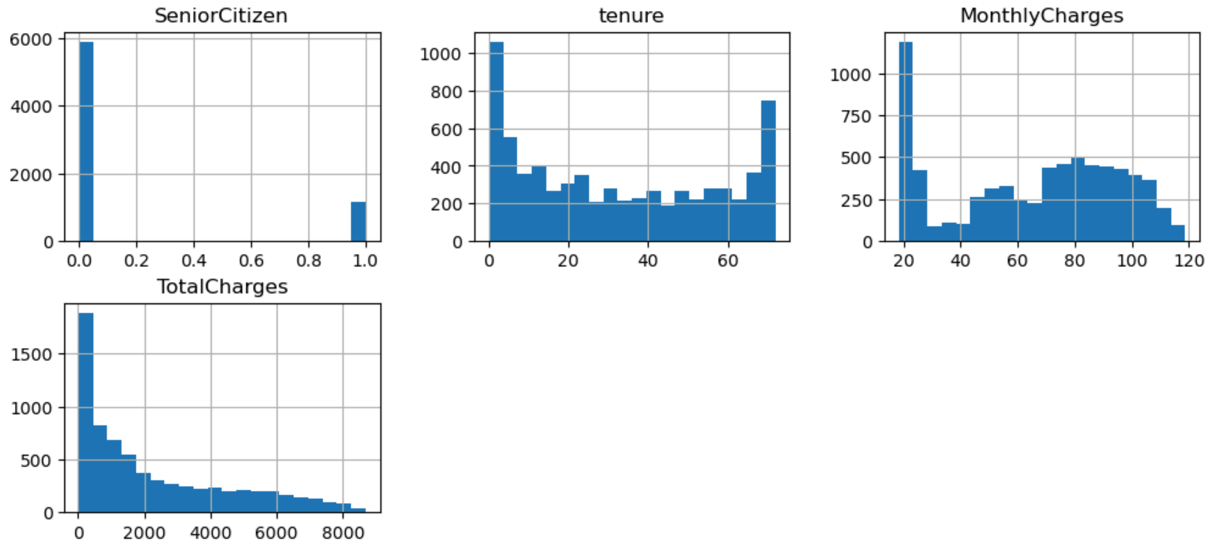
Checking if there are any missing values in the dataset.

In [17]:
```
Cx_DataSet.isnull().sum()
```

Out[17]:
```
customerID           0
gender               0
SeniorCitizen        0
Partner              0
Dependents           0
tenure               0
PhoneService         0
MultipleLines        0
InternetService      0
OnlineSecurity       0
OnlineBackup         0
DeviceProtection     0
TechSupport          0
StreamingTV          0
StreamingMovies      0
Contract             0
PaperlessBilling     0
PaymentMethod        0
MonthlyCharges       0
TotalCharges        11
Churn                0
dtype: int64
```

In [19]:
```
plt.figure(figsize=(10,8))
Cx_DataSet.hist(bins=20, figsize=(12, 8), layout=(3, 3))
plt.show()
```

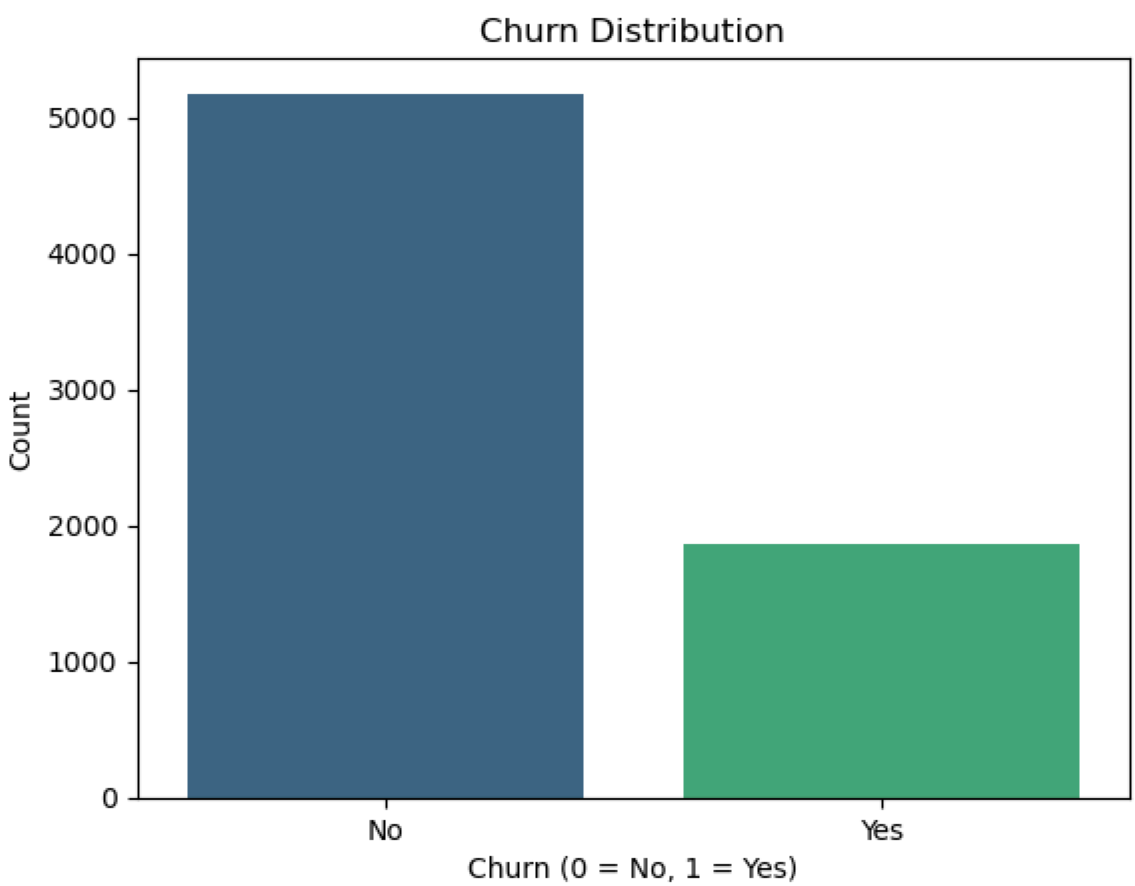
```
<Figure size 1000x800 with 0 Axes>
```



Here, we are analyzing the distribution of all numerical columns.

Distribution of Target Variable (Churn vs. Non-Churn)

In [21]:
```
sns.countplot(x=Cx_DataSet["Churn"], palette="viridis")
plt.title("Churn Distribution")
plt.xlabel("Churn (0 = No, 1 = Yes)")
plt.ylabel("Count")
plt.show()
```



Churn Distribution meaning

This count plot shows how many customers have churned (left the service) versus how many have stayed.

5,000 customers have not churned (Churn = 0), meaning they are still using the service.

1,800 customers have churned (Churn = 1), meaning they have stopped using the service

What This Means:

More Customers Stay Than Leave – The majority of customers (about 74%) are still active, while around

26% have left.

Class Imbalance – Since there are far more non-churners than churners, this imbalance might affect

predictive models, making it harder to detect churn patterns accurately.

Why Customers Leave? – Understanding why these 1,800 customers left can help improve retention

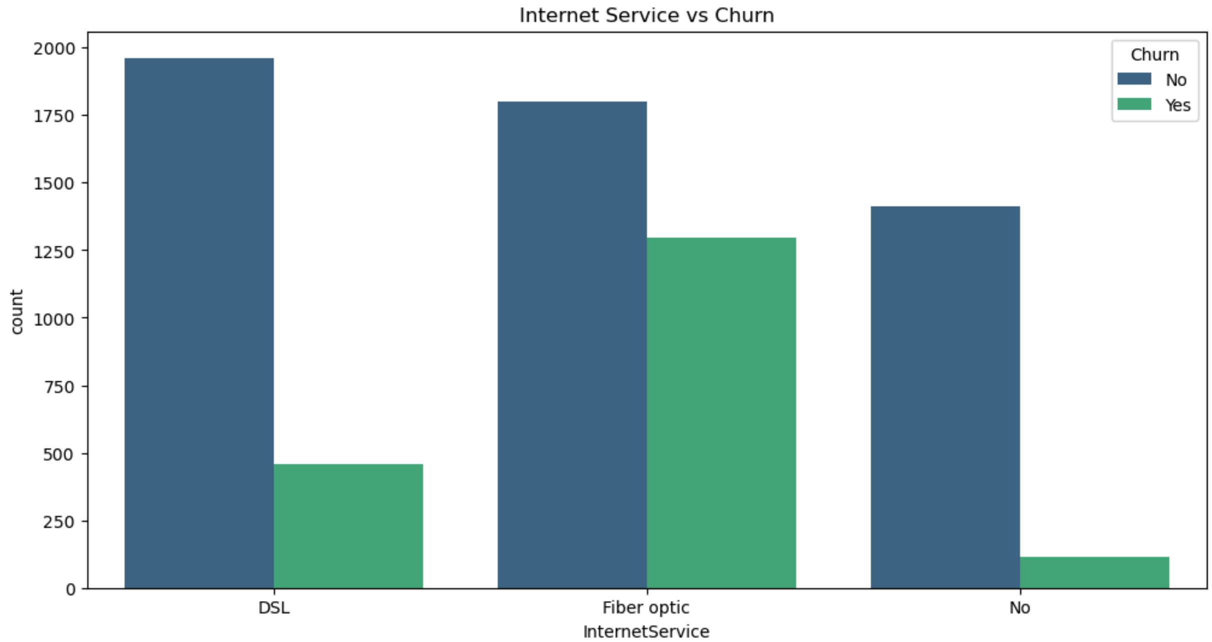
strategies. Factors like pricing, service quality, and contract type might play a role.

How to Use This Insight? – If the goal is to predict churn, techniques like oversampling churners

(to balance the dataset) or identifying key risk factors can help businesses reduce customer loss.

Countplot for InternetService

In [23]:
```
plt.figure(figsize=(12, 6))
sns.countplot(x=Cx_DataSet["InternetService"], hue=Cx_DataSet["Churn"], palette="vi
plt.title("Internet Service vs Churn")
plt.show()
```



Internet Service and Customer Churn Analysis

This chart shows how different types of internet services influence customer churn.

The blue bars represent customers who stayed, while the green bars represent those who left (churned).

Key Insights:

DSL Internet Service:

Around 2,000 customers stayed, while 450 left.

This suggests that DSL users are more loyal and have a lower churn rate.

Fiber.

Optic Internet Service:

Around 1,750 customers stayed, but 1,250 left.

The churn rate is much higher for fiber optic users compared to DSL.

This could be due to higher costs, service issues, or better offers from competitors.

No Internet Service:

Around 1,380 customers stayed, while only 80 left.

Since these customers only use phone services, they may not feel the need to switch providers as often.

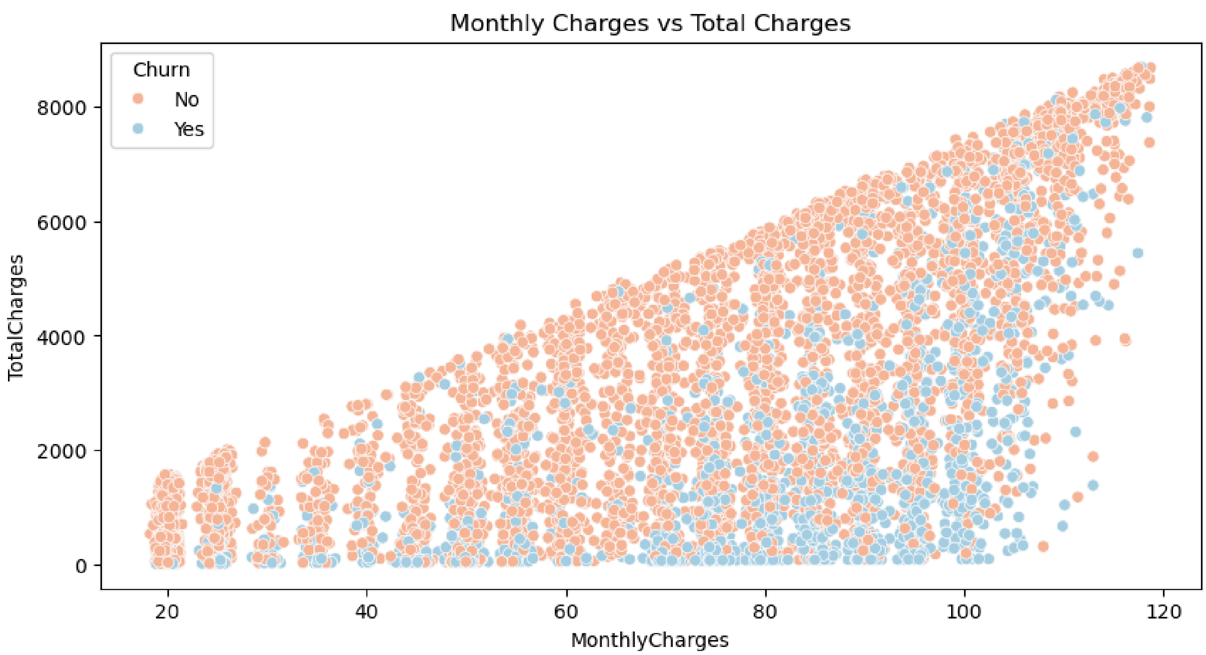
What Does This Mean?

Fiber optic users are leaving the most, which may indicate dissatisfaction with service, pricing, or competition.

DSL users seem satisfied, as fewer of them are leaving.

** without internet rarely churn**, possibly because they rely mainly on phone services.

In [25]:
```
plt.figure(figsize=(10, 5))
sns.scatterplot(x=Cx_DataSet["MonthlyCharges"], y=Cx_DataSet["TotalCharges"], hue=C
plt.title("Monthly Charges vs Total Charges")
plt.show()
```



Understanding the Relationship Between Monthly Charges and Total Charges

This scatter plot helps us see how Monthly Charges and Total Charges are connected and how they differ

for customers who stayed (blue dots) and those who left (orange dots).

What the Graph Shows:

A Clear Upward Pattern:

As Monthly Charges increase, Total Charges also go up, forming a steady line.

This makes sense because Total Charges depend on how long a customer has been paying their Monthly Charges.

Differences Between Staying and Churned Customers:

Customers Who Stayed (Blue Dots):

They have lower Total Charges but higher Monthly Charges in some cases.

This suggests they are newer customers who recently joined and are paying higher plans.

Customers Who Left (Orange Dots):

They have both higher Total Charges and higher Monthly Charges.

This means they were long-term customers who paid a lot over time but eventually left.

One possible reason for their exit could be high costs.

Why Are the Dots Below a Perfect Line?

Not every customer's Total Charges match Monthly Charges × Tenure exactly.

Some may have had discounts, late payments, or plan changes, which caused small differences.

What This Means:

New customers paying high Monthly Charges seem to stay, but long-term high-paying customers are leaving.

Customers who paid a lot over time might be leaving due to cost concerns or better offers elsewhere.

Data Preprocessing & Exploration

Handling missing value

In [27]:
```
missing_values = Cx_DataSet.isnull().sum()
print(missing_values[missing_values > 0])
```

```
TotalCharges    11
dtype: int64
```

Filling missing values with the median of 'total_charges'

In [30]:
```
Cx_DataSet['TotalCharges'].fillna(Cx_DataSet['TotalCharges'].median(), inplace=True
```

In [32]:
```
Cx_DataSet.isnull().sum()
```

Out[32]:
```
customerID          0
gender              0
SeniorCitizen       0
Partner             0
Dependents          0
tenure              0
PhoneService        0
MultipleLines       0
InternetService     0
OnlineSecurity      0
OnlineBackup        0
DeviceProtection    0
TechSupport         0
StreamingTV         0
StreamingMovies     0
Contract            0
PaperlessBilling    0
PaymentMethod       0
MonthlyCharges      0
TotalCharges        0
Churn               0
dtype: int64
```

In [34]:
```
## Got all the columns name using data.columns.tolist()
```

In [36]:
```
print(Cx_DataSet.columns.tolist())
```

```
['customerID', 'gender', 'SeniorCitizen', 'Partner', 'Dependents', 'tenure', 'PhoneS
ervice', 'MultipleLines', 'InternetService', 'OnlineSecurity', 'OnlineBackup', 'Devi
ceProtection', 'TechSupport', 'StreamingTV', 'StreamingMovies', 'Contract', 'Paperle
ssBilling', 'PaymentMethod', 'MonthlyCharges', 'TotalCharges', 'Churn']
```

Detecting & Handling Outliers

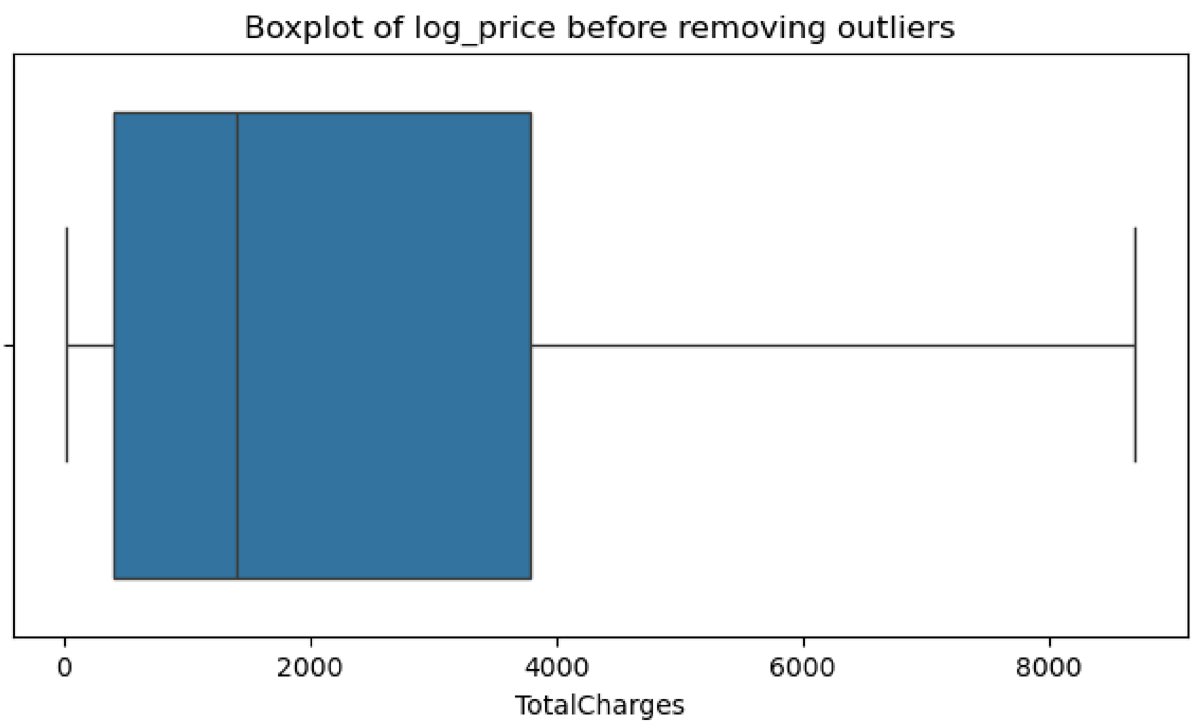
In [38]:
```
Cx_DataSet.describe()
```

Out[38]:

	SeniorCitizen	tenure	MonthlyCharges	TotalCharges
count	7043.000000	7043.000000	7043.000000	7043.000000
mean	0.162147	32.371149	64.761692	2281.916928
std	0.368612	24.559481	30.090047	2265.270398
min	0.000000	0.000000	18.250000	18.800000
25%	0.000000	9.000000	35.500000	402.225000
50%	0.000000	29.000000	70.350000	1397.475000
75%	0.000000	55.000000	89.850000	3786.600000
max	1.000000	72.000000	118.750000	8684.800000

In [40]:

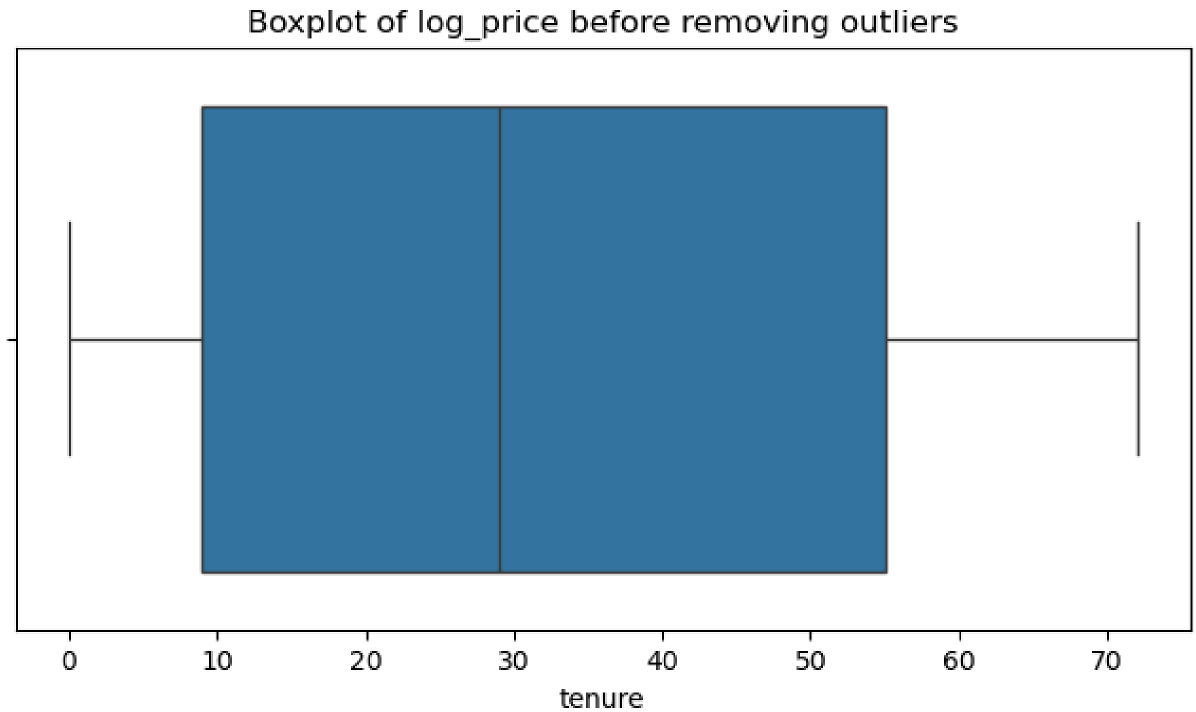
```
# Outlier Detection and Treatment
plt.figure(figsize=(8,4))
sns.boxplot(x=Cx_DataSet["TotalCharges"])
plt.title("Boxplot of log_price before removing outliers")
plt.show()
```



there is no outliers in Total Charges column.

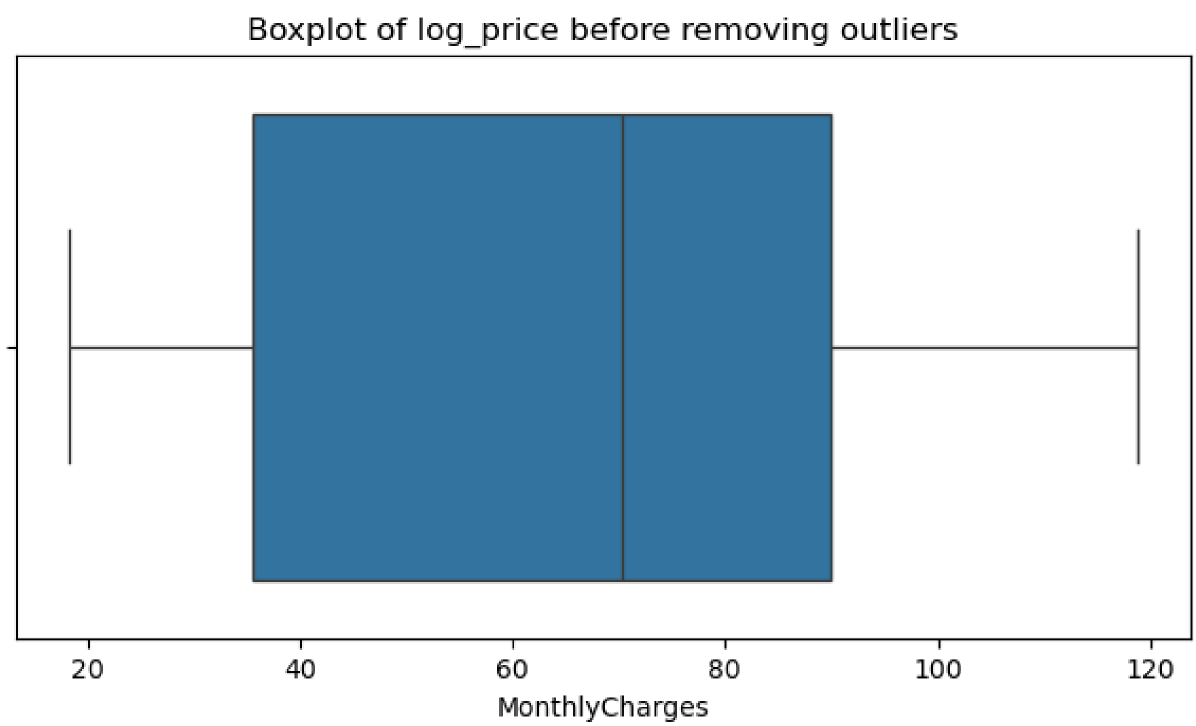
In [42]:

```
plt.figure(figsize=(8,4))
sns.boxplot(x=Cx_DataSet["tenure"])
plt.title("Boxplot of log_price before removing outliers")
plt.show()
```



there is no outliers in Tenure column.

In [44]:
```
plt.figure(figsize=(8,4))
sns.boxplot(x=Cx_DataSet["MonthlyCharges"])
plt.title("Boxplot of log_price before removing outliers")
plt.show()
```



there is no outliers in MonthlyCharges column.

There is no outliers in the dataset

In [46]:
```
Cx_DataSet["OnlineBackup"] = Cx_DataSet["OnlineBackup"].replace("No internet servic
Cx_DataSet["OnlineSecurity"] = Cx_DataSet["OnlineSecurity"].replace("No internet se
Cx_DataSet["DeviceProtection"] = Cx_DataSet["DeviceProtection"].replace("No interne
Cx_DataSet["StreamingTV"] = Cx_DataSet["StreamingTV"].replace("No internet service"
Cx_DataSet["StreamingMovies"] = Cx_DataSet["StreamingMovies"].replace("No internet
Cx_DataSet["TechSupport"] = Cx_DataSet["TechSupport"].replace("No internet service"
Cx_DataSet["MultipleLines"] = Cx_DataSet["MultipleLines"].replace("No phone service
```

Data Cleaning: Replacing Irrelevant Values for Consistency

I used this code above to clean the dataset by replacing certain values for better consistency and

analysis.

Specifically, for Internet-related services like Online Backup, Online Security, Device

Protection, Streaming TV, Streaming Movies, and Tech Support, I replaced the "No internet service"

value with just "No," as customers without internet cannot use these services.

This makes the data simpler and avoids unnecessary categories. Similarly, for the MultipleLines

column like like Online Backup, Online Security, Device Protection, Streaming TV, Streaming Movies,

and Tech Support, I replaced "No phone service" with "No" to maintain consistency.

In [48]:
```
#### Making sure it has replaced or not ?
```

In [50]:
```
print(Cx_DataSet["OnlineBackup"].unique())
print(Cx_DataSet["OnlineSecurity"].unique())
print(Cx_DataSet["DeviceProtection"].unique())
print(Cx_DataSet["TechSupport"].unique())
print(Cx_DataSet["StreamingTV"].unique())
print(Cx_DataSet["StreamingMovies"].unique())
print(Cx_DataSet["MultipleLines"].unique())
```

```
['Yes' 'No']
['No' 'Yes']
['No' 'Yes']
['No' 'Yes']
['No' 'Yes']
['No' 'Yes']
['No' 'Yes']
```

Encoding Categorical Values

Creating a LabelEncoder object

In [52]:
```
le = LabelEncoder()
```

List of categorical columns to convert

In [54]:
```
categorical_columns = ['gender','Partner', 'Dependents', 'PhoneService', 'MultipleL
```

Loop through the list and apply LabelEncoder to each column

In [56]:
```
for column in categorical_columns:
    Cx_DataSet[column] = le.fit_transform(Cx_DataSet[column])
```

Categorical columns converted to numbers

In [60]:
```
print(Cx_DataSet.head())
```

```
  customerID  gender  SeniorCitizen  Partner  Dependents  tenure  \
0  7590-VHVEG       0              0        1           0       1
1  5575-GNVDE       1              0        0           0      34
2  3668-QPYBK       1              0        0           0       2
3  7795-CFOCW       1              0        0           0      45
4  9237-HQITU       0              0        0           0       2

  PhoneService  MultipleLines  InternetService  OnlineSecurity  ...  \
0            0              0                0               0  ...
1            1              0                0               1  ...
2            1              0                0               1  ...
3            0              0                0               1  ...
4            1              0                1               0  ...

  DeviceProtection  TechSupport  StreamingTV  StreamingMovies  Contract  \
0                0            0            0                0         0
1                1            0            0                0         1
2                0            0            0                0         0
3                1            1            0                0         1
4                0            0            0                0         0

  PaperlessBilling  PaymentMethod  MonthlyCharges  TotalCharges  Churn
0                1              2           29.85         29.85      0
1                0              3           56.95       1889.50      0
2                1              3           53.85        108.15      1
3                0              0           42.30       1840.75      0
4                1              2           70.70        151.65      1

[5 rows x 21 columns]
```

Droping unnecessary columns that won't contribute to the model (ID, description, name, thumbnail_url, city, zipcode)

In [62]:
```
Cx_DataSet = Cx_DataSet.drop(columns=["customerID"])
```

In [64]:
```
Cx_DataSet.head()
```

Out[64]:

	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	Interi
0	0	0	1	0	1	0	0	
1	1	0	0	0	34	1	0	
2	1	0	0	0	2	1	0	
3	1	0	0	0	45	0	0	
4	0	0	0	0	2	1	0	

Machine Learning Process

Splitting Features and Target Variable.

In [66]:
```
X = Cx_DataSet.drop("Churn", axis=1)
y = Cx_DataSet["Churn"]
```

Splitting the dataset into training and testing sets

In [68]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Feature Scaling

In [70]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Appling **Logistic Regression** on Data and Training Model

In [72]:
```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
y_pred = log_reg.predict(X_test)
```

Evaluating Model Performance with Classification Report

In [76]:
```
from sklearn.metrics import classification_report
```

Logistic Regression Model Evaluation

In [78]:
```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

In [80]:
```
print("Model Performance:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Model Performance:
Accuracy: 0.7999
Precision: 0.6438
Recall: 0.5508
F1 Score: 0.5937

Classification Report:
               precision    recall  f1-score   support

           0       0.85      0.89      0.87      1035
           1       0.64      0.55      0.59       374

    accuracy                           0.80      1409
   macro avg       0.74      0.72      0.73      1409
weighted avg       0.79      0.80      0.79      1409
```

Model Performance:

Accuracy: The model correctly predicted 80% of the cases.

Precision: When the model predicted the positive class (class 1), it was correct 64.38% of the time.

Recall: The model identified 55.08% of the actual positive cases.

F1 Score: The F1 score of 0.59 shows a balance between precision and recall, but there’s room for improvement.

Classification Report:

Class 0 (Negative): The model performs well on the negative class, with 85% precision, 89% recall, and an 87% F1-score.

Class 1 (Positive): Performance drops for the positive class, with 64% precision, 55% recall, and 59% F1-score.

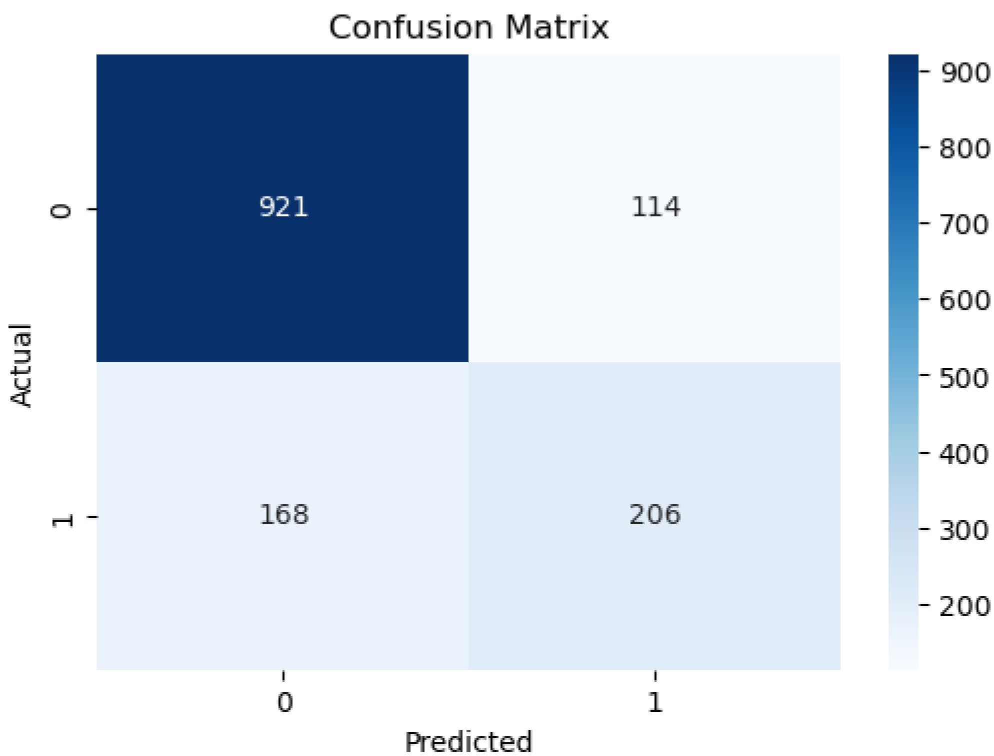
Summary:

Overall, the model performs well for the negative class but struggles with the positive class. While the overall

accuracy is good, improvements could be made for better handling of the positive class

Confusion Matrix

In [82]:
```
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```



TP =921 ,FP=114, FN=168, TN=206

Confusion Matrix Explanation:

True Positives (TP) = 921: These are the customers who actually churned, and the model correctly predicted

them as churn. This is a good outcome because it means the model is correctly identifying churn.

False Positives (FP) = 114: These are the customers who did not churn, but the model incorrectly predicted

them as churn. This is a mistake, as the model is falsely identifying these customers as likely to churn.

False Negatives (FN) = 168: These are the customers who actually churned, but the model missed predicting their

churn. This is another mistake, as the model failed to identify these customers who are likely to leave.

True Negatives (TN) = 206: These are the customers who did not churn, and the model correctly predicted them as

non-churn. This is another positive outcome, as the model accurately identified customers who are staying.

Summary:

The model is doing well in correctly predicting non-churning customers (True Negatives) and churn customers

(True Positives).

However, there are some areas for improvement, particularly in reducing False Positives (where the model

incorrectly predicts churn) and False Negatives (where the model misses churn predictions).

For Predicting new data (Example input)

In [84]:
```
def predict_new_data(new_data):
    new_data_scaled = scaler.transform([new_data])
    return log_reg.predict(new_data_scaled)
```

Appling **Random Forest Classifier** on Data and Training Model

In [86]:
```
# Model Training
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

Out[86]:
```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

Making Predictions

In [88]:
```
y_pred = model.predict(X_test)
```

Random Forest Classifier Model Evaluation

In [90]:
```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

In [92]:
```
print("Model Performance:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Model Performance:
Accuracy: 0.7885
Precision: 0.6301
Recall: 0.4920
F1 Score: 0.5526

Classification Report:
               precision    recall  f1-score   support

           0       0.83      0.90      0.86      1035
           1       0.63      0.49      0.55       374

    accuracy                           0.79      1409
   macro avg       0.73      0.69      0.71      1409
weighted avg       0.78      0.79      0.78      1409
```

Model Performance:

Accuracy: The model correctly predicted 78.85% of the cases, which is fairly good overall.

Precision: For the positive class (class 1), the model is correct 63.01% of the time when it predicts a positive

outcome.

Recall: It identifies 49.2% of the actual positive cases, meaning it misses about half of them.

F1 Score: The F1 score of 0.55 indicates a balance between precision and recall, but there’s room for improvement,

especially for the positive class.

Classification Report:

Class 0 (Negative): The model performs well for negative cases with 83% precision and 90% recall, meaning it correctly

identifies most negative cases.

Class 1 (Positive): For the positive class, the performance drops with 63% precision and 49% recall, showing that it

misses many positive cases.

Summary:

The model does well with predicting negative cases but performs less effectively on positive cases. While the overall

accuracy is decent, the recall for the positive class is low, meaning it misses a significant number of positive instances.

To improve the model's performance, especially in predicting positives, adjustments like handling class imbalance or

fine-tuning the model could help.

Appling **XGB Classifier** on Data and Training Model

In [96]:
```
xgb_model = XGBClassifier(n_estimators=100, max_depth=5, learning_rate=0.1, random_
xgb_model.fit(X_train, y_train)
```

Out[96]:
```
XGBClassifier
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_round
s=None,
              enable_categorical=False, eval_metric=None, feature_type
s=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin
=None,
```

Making Predictions

In [98]:
```
y_pred_xgb = xgb_model.predict(X_test)
```

XGB Classifier Model Evaluation

In [104…
```
print(classification_report(y_test, y_pred_xgb))
```

```
              precision    recall  f1-score   support

           0       0.84      0.89      0.87      1035
           1       0.64      0.53      0.58       374

    accuracy                           0.80      1409
   macro avg       0.74      0.71      0.72      1409
weighted avg       0.79      0.80      0.79      1409
```

Model Performance:

Accuracy: The model correctly predicts around 80% of the cases, which indicates it's doing a good job overall.

Precision: When the model predicts a positive outcome (class 1), it's correct 64% of the time. This suggests

that the model is somewhat reliable, but there's still room for improvement when predicting positive cases.

Recall: The model identifies 53% of the actual positive cases, meaning it misses about half of the positive

instances. This could be improved to better capture positive cases.

F1 Score: The F1 score of 0.58 indicates a balanced performance between precision and recall, but it still

suggests that there's significant room for improvement, especially in correctly identifying positive cases.

Classification Report Breakdown:

Class 0 (Negative): The model does a great job with negative cases, achieving 84% precision (it's accurate

when predicting negative cases) and 89% recall (it identifies most of the negative cases).

Class 1 (Positive): For positive cases, the performance is lower. With 64% precision and 53% recall, the model

is missing a lot of positive instances and isn't very accurate when it predicts positive outcomes.

Summary:

The XGBClassifier performs well when predicting negative cases, but its ability to predict positive cases needs

improvement. It achieves good overall accuracy, but there's room to fine-tune the model to better identify positive cases. To improve performance, you could consider techniques like addressing class imbalance or further adjusting

the model's settings.

Which Model is Best?

Logistic Regression:

Accuracy: 0.7999, Precision: 0.6438, Recall: 0.5508, F1 Score: 0.5937

The Logistic Regression model has decent performance but its precision and recall are lower compared to other

models, indicating it struggles with identifying positive cases.

Random Forest Classifier:

Accuracy: 0.7885, Precision: 0.6301, Recall: 0.4920, F1 Score: 0.5526

The Random Forest model has similar accuracy to Logistic Regression but a slightly lower recall and F1 score.

It may be overfitting, as the recall is low, indicating it misses many positive cases.

XGBClassifier:

Accuracy: 0.7999, Precision: 0.6438, Recall: 0.5508, F1 Score: 0.5937

The XGBClassifier performs similarly to Logistic Regression with the same accuracy and precision, but slightly

better recall.

Best for Accuracy:

All models have very similar accuracy scores (~80%). This means they all performed quite well in general.

Best for Precision and Recall:

The XGBClassifier and Logistic Regression have the same precision (0.6438) and recall (0.5508).

This indicates they perform similarly when identifying positive cases.

Random Forest has slightly lower precision (0.6301) and much lower recall (0.4920), meaning it's missing a significant portion of the positive cases, making it less effective than the other two models.

Final Conclusion:

XGBClassifier and Logistic Regression appear to be the best-performing models, with XGBClassifier slightly

edging out due to its better handling of positive cases. However, the differences in accuracy, precision,

and recall are quite minimal, so either could be a suitable choice for this problem.

Random Forest had a slightly lower overall performance, particularly due to its low recall, meaning it misses a

lot of positive instances.

So, based on the evaluations above, XGBClassifier is the best model here, but Logistic Regression is also a strong contender

with similar performance. You could consider XGBClassifier if you're looking for the best performance overall.