1. Setup and Data Loading

In [301…

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, accuracy_sc
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV

from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier


plt.style.use('ggplot')

# Load dataset
telco = pd.read_csv("Telco-Dataset.csv")
```

In [302…

```
telco.head()
```

Out[302…

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	Mul
0	7590-VHVEG	Female	0	Yes	No	1	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	
4	9237-HQITU	Female	0	No	No	2	Yes	

5 rows × 21 columns

2. Data Exploration

2.1. Explaining the Dataset

What is this dataset about? This dataset contains information about 7,043 customers from a telecommunications company. It tracks customer demographics (age, family

status), account information (how long they have been a customer, how they pay), and the specific services they have signed up for (internet, phone, tech support, etc.).

What are we going to do with it? We are going to build a **Binary Classification** Machine Learning model. Our primary goal is to predict the `Churn` column—whether a customer left the company within the last month (Yes or No). By identifying the mathematical patterns of customers who leave (e.g., month-to-month contracts, high monthly charges, lack of tech support), we can help the business proactively identify at-risk customers and offer them incentives to stay, directly saving the company money.

Data Dictionary Below is a breakdown of every feature available to our model:

Column Name	Description
`customerID`	Unique alphanumeric identifier for each customer.
`gender`	Whether the customer is a male or a female.
`SeniorCitizen`	Whether the customer is a senior citizen or not (1 = Yes, 0 = No).
`Partner`	Whether the customer has a partner or not (Yes, No).
`Dependents`	Whether the customer has dependents/children or not (Yes, No).
`tenure`	Number of months the customer has stayed with the company.
`PhoneService`	Whether the customer has a phone service or not (Yes, No).
`MultipleLines`	Whether the customer has multiple lines (Yes, No, No phone service).
`InternetService`	Customer's internet service type (DSL, Fiber optic, No).
`OnlineSecurity`	Whether the customer has online security add-on (Yes, No, No internet).
`OnlineBackup`	Whether the customer has online backup add-on (Yes, No, No internet).
`DeviceProtection`	Whether the customer has device protection add-on (Yes, No, No internet).
`TechSupport`	Whether the customer has tech support add-on (Yes, No, No internet).
`StreamingTV`	Whether the customer uses the internet to stream TV (Yes, No, No internet).
`StreamingMovies`	Whether the customer uses the internet to stream movies (Yes, No, No internet).
`Contract`	The contract term of the customer (Month-to-month, One year, Two year).
`PaperlessBilling`	Whether the customer uses paperless billing (Yes, No).
`PaymentMethod`	How the customer pays (Electronic check, Mailed check, Bank transfer, Credit card).
`MonthlyCharges`	The amount charged to the customer every month.
`TotalCharges`	The total cumulative amount charged to the customer.

Column Name	Description
`Churn`	**Target Variable:** Did the customer leave within the last month? (Yes, No).

2.2. Dataset Info

In [303…
```
telco.info()
```

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

We have a pretty good dataset ! ***No NaN values !***

Though, the `TotalCharges` column is in str type so we should not forget to convert it to a numeric type !

In [304…
```
telco.describe()
```

Out[304...

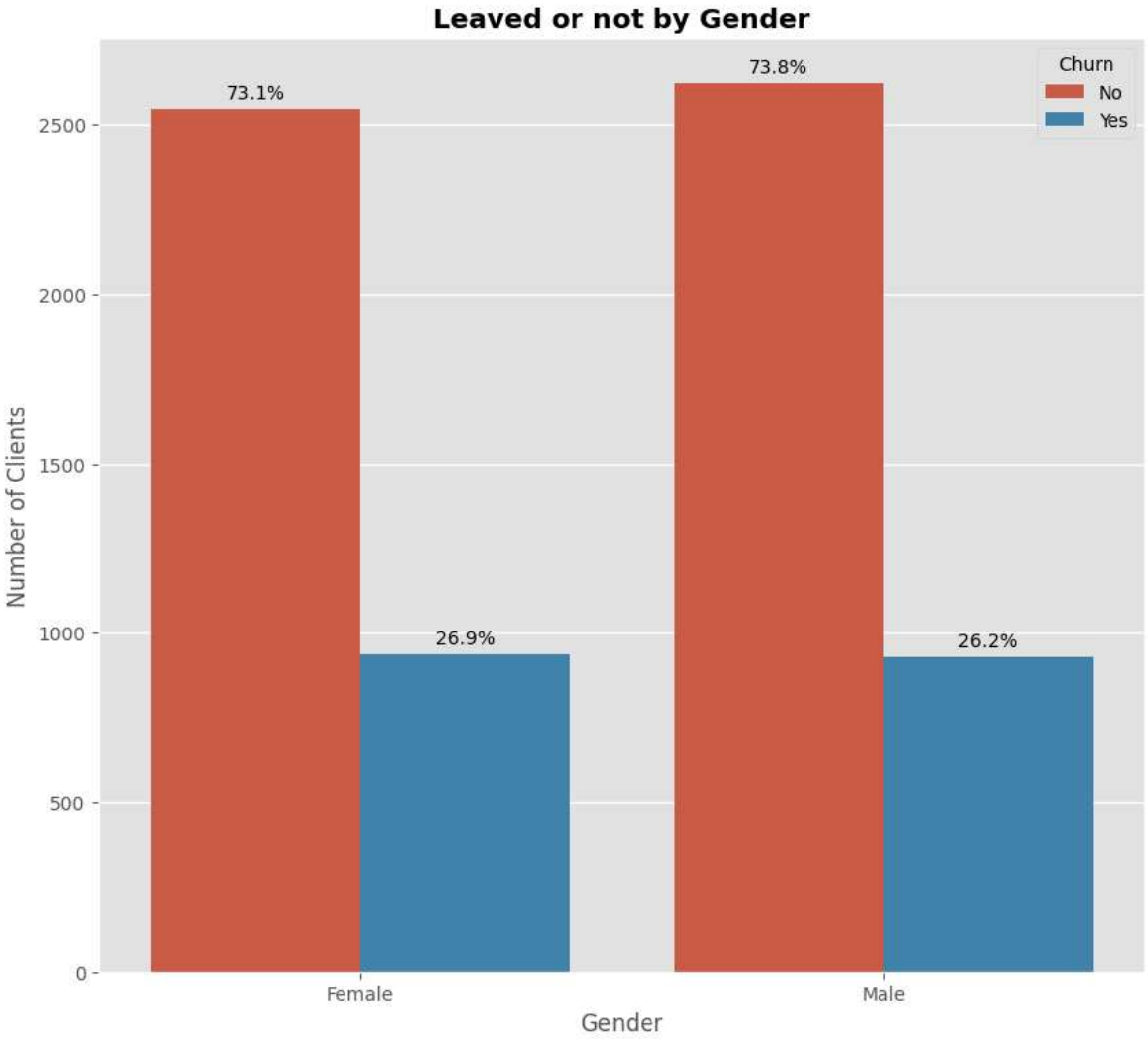
	SeniorCitizen	tenure	MonthlyCharges
count	7043.000000	7043.000000	7043.000000
mean	0.162147	32.371149	64.761692
std	0.368612	24.559481	30.090047
min	0.000000	0.000000	18.250000
25%	0.000000	9.000000	35.500000
50%	0.000000	29.000000	70.350000
75%	0.000000	55.000000	89.850000
max	1.000000	72.000000	118.750000

3. Exploratory Data Analysis

3.1. Which Sex quits more ?

In [305...

```
plt.figure(figsize=(10, 9))

ax = sns.countplot(telco, x="gender", hue="Churn")
plt.title('Leaved or not by Gender', fontweight='bold')
plt.xlabel('Gender')
plt.ylabel('Number of Clients')

gender_totals = telco['gender'].value_counts()

for c in ax.containers:
    labels = []
    for i, bar in enumerate(c):
        gender_name = ax.get_xticklabels()[i].get_text()
        percentage = (bar.get_height() / gender_totals[gender_name]) * 100
        labels.append(f'{percentage:.1f}%')
    ax.bar_label(c, labels=labels, padding=3)

plt.show()
```



The visualization reveals that the churn rate is nearly identical between male and female customers.

This is an expected outcome. Telecommunication services (internet, phone lines) are universal utilities. Unlike highly segmented industries (such as specialized apparel or cosmetics), a customer's decision to cancel a basic utility contract is driven by service quality, pricing, and customer support, rather than demographic factors like gender.

Gender is Noise

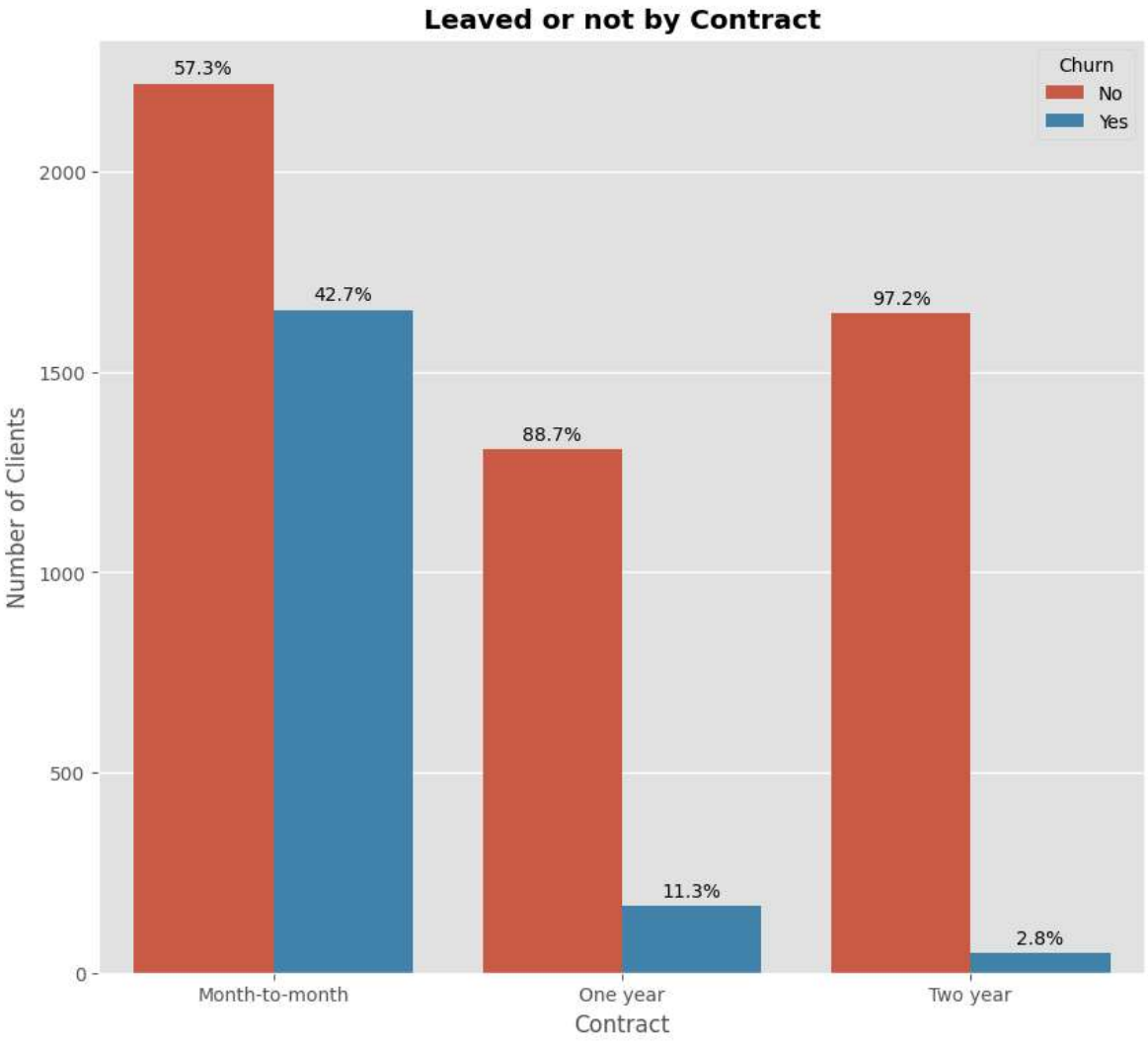
3.2. Which Contract affects `Churn` ?

In [306…
```
telco['Contract'].value_counts()
```

Out[306…
```
Contract
Month-to-month    3875
Two year          1695
One year          1473
Name: count, dtype: int64
```

In [307…
```
plt.figure(figsize=(10, 9))

ax = sns.countplot(telco, x="Contract", hue="Churn")
plt.title('Leaved or not by Contract', fontweight='bold')
```

```
plt.xlabel('Contract')
plt.ylabel('Number of Clients')

contract_totals = telco['Contract'].value_counts()

for c in ax.containers:
    labels = []
    for i, bar in enumerate(c):
        contract_name = ax.get_xticklabels()[i].get_text()
        percentage = (bar.get_height() / contract_totals[contract_name]) * 100
        labels.append(f'{percentage:.1f}%')
    ax.bar_label(c, labels=labels, padding=3)

plt.show()
```



📊 Insight

Observation: The visualization clearly demonstrates that the length of a customer's contract is heavily tied to their likelihood of leaving.

- **Month-to-month** customers have an alarmingly high churn rate of **42.7%.**
- **One-year** contracts see a massive drop in churn, falling to just **11.3%.**
- **Two-year** contracts are highly secure, with a churn rate of only **2.8%.**

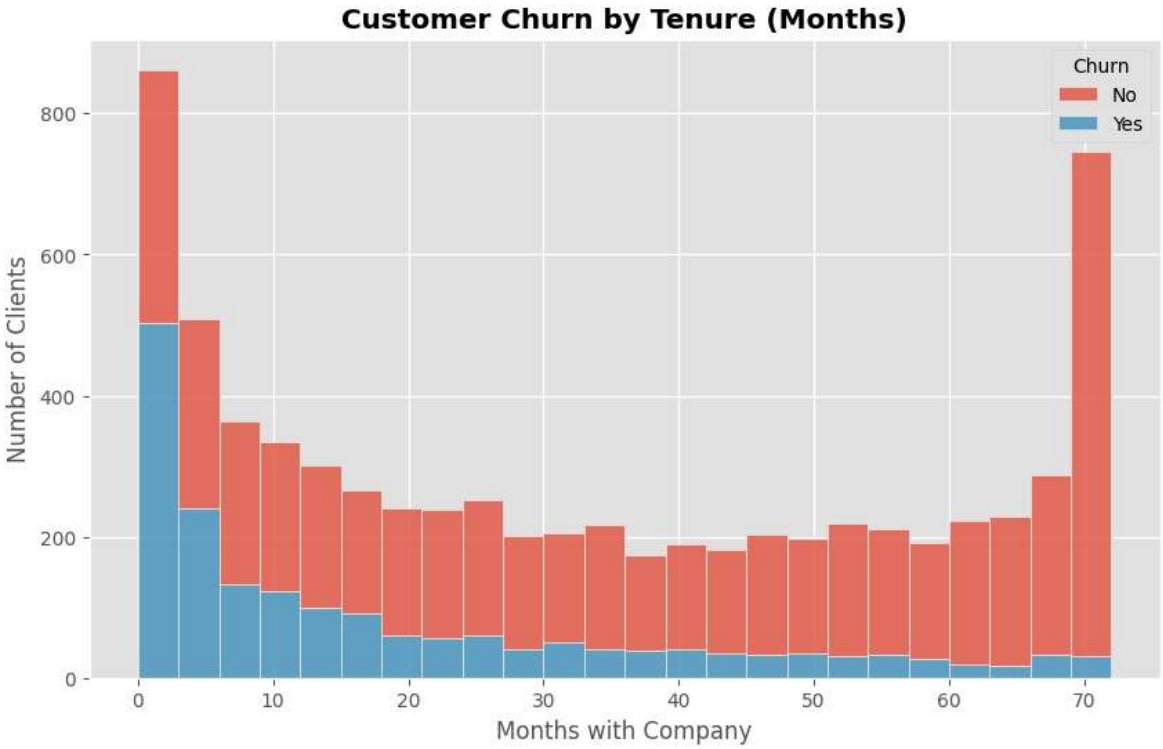
Business Context: This makes perfect logical sense. Month-to-month contracts offer maximum flexibility, meaning customers can easily leave for a competitor's promotional

offer without facing cancellation fees. Customers locked into 1-year or 2-year contracts are heavily disincentivized from leaving early.

3.3. Does `tenure` affects `Churn` ?

In [308…

```
plt.figure(figsize=(10, 6))

sns.histplot(data=telco, x="tenure", hue="Churn", multiple="stack", bins=24)

plt.title('Customer Churn by Tenure (Months)', fontweight='bold')
plt.xlabel('Months with Company')
plt.ylabel('Number of Clients')

plt.show()
```



📊 Insight

This histogram perfectly illustrates that the absolute highest risk of churn occurs right at the beginning of the customer journey. We see a massive spike in cancellations during the first few months, which then drops off a cliff and stabilizes as time goes on. Conversely, there is a huge cluster of fiercely loyal customers at the 72-month mark who almost never leave. This tells the business exactly where to focus their retention budget: improving the new-customer onboarding experience. If the company can successfully keep a user happy for just the first 6 to 12 months, that user is highly likely to stay for years.

3.4. Does `MonthlyCharges` affects `Churn` ?

In [309…

```
telco['MonthlyCharges'].value_counts()
```

Out[309…
```
MonthlyCharges
20.05     61
19.85     45
19.95     44
19.90     44
19.70     43
          ..
72.00      1
108.35     1
63.10      1
44.20      1
78.70      1
Name: count, Length: 1585, dtype: int64
```

In [310…
```
telco['MonthlyCharges'].describe()
```

Out[310…
```
count    7043.000000
mean       64.761692
std        30.090047
min        18.250000
25%        35.500000
50%        70.350000
75%        89.850000
max       118.750000
Name: MonthlyCharges, dtype: float64
```

The `MonthlyCharges` column contains hundreds of unique, continuous numerical values. That's why we will apply a technique called **Quantile Binning**. This converts the exact prices into discrete, easy-to-understand categorical tiers.

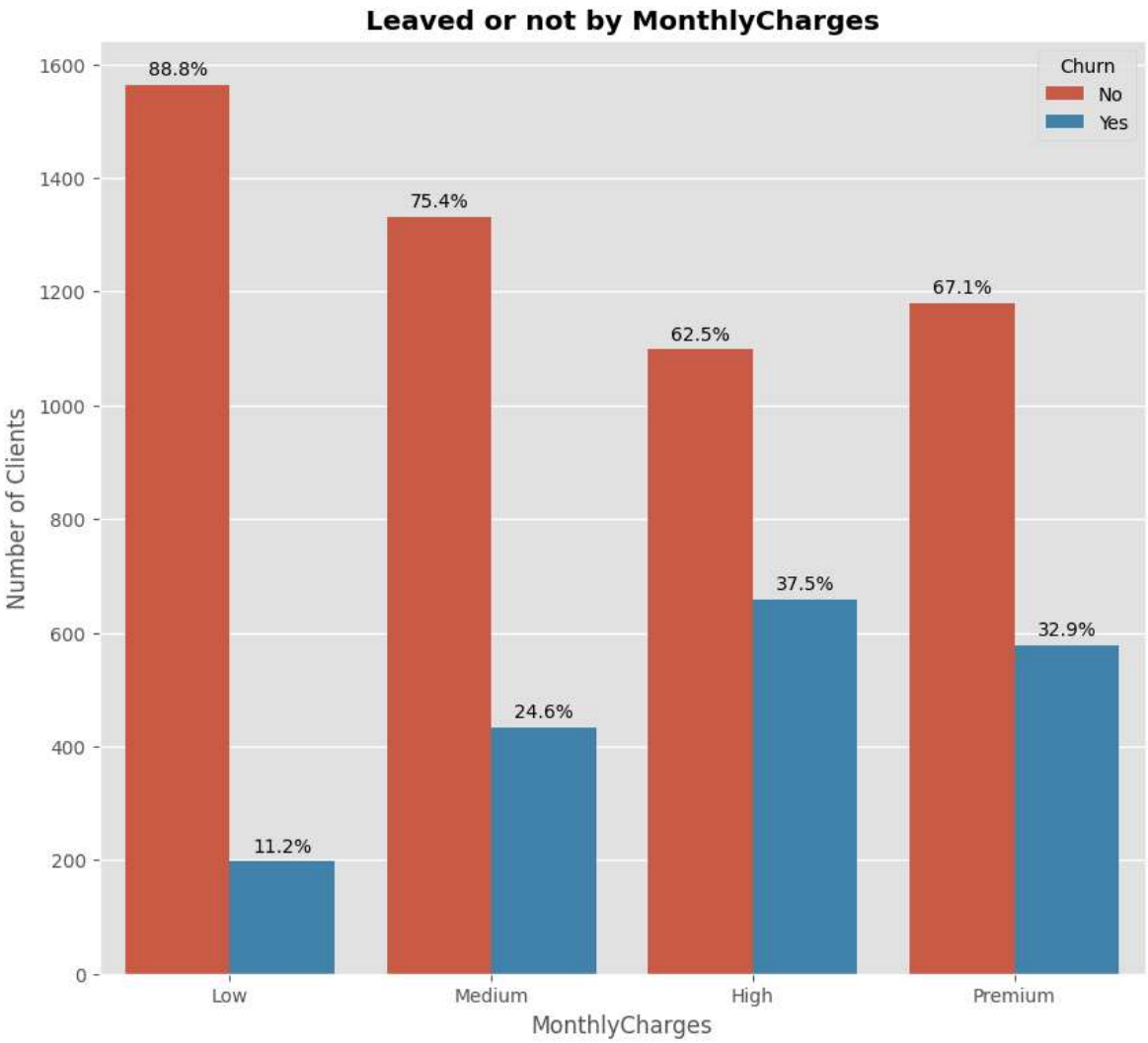
The Tiers: Using the dataset's statistical percentiles, we will classify the monthly charges into the following categories:

Price Category	Percentile Range	Charge Threshold
Low	0% - 25%	<= $35.50
Medium	25% - 50%	35.51—70.35
High	50% - 75%	70.36—89.85
Premium	75% - 100%	> $89.85

In [311…
```
telco['MonthlyCharges_categorial'] = pd.qcut(telco['MonthlyCharges'],
                                q=4,
                                labels=['Low', 'Medium', 'High', 'Premium'],
                                retbins=False,
                                precision=3,
                                duplicates='raise')

telco['MonthlyCharges_categorial'].head()
```

Out[311…
```
0       Low
1    Medium
2    Medium
3    Medium
4      High
Name: MonthlyCharges_categorial, dtype: category
Categories (4, str): ['Low' < 'Medium' < 'High' < 'Premium']
```

In [312…
```
plt.figure(figsize=(10, 9))

ax = sns.countplot(telco, x="MonthlyCharges_categorial", hue="Churn")
plt.title('Leaved or not by MonthlyCharges', fontweight='bold')
plt.xlabel('MonthlyCharges')
plt.ylabel('Number of Clients')

mcharges_totals = telco['MonthlyCharges_categorial'].value_counts()

for c in ax.containers:
    labels = []
    for i, bar in enumerate(c):
        mcharges_name = ax.get_xticklabels()[i].get_text()
        percentage = (bar.get_height() / mcharges_totals[mcharges_name]) * 100
        labels.append(f'{percentage:.1f}%')
    ax.bar_label(c, labels=labels, padding=3)

plt.show()
```



📊 Insight

The chart reveals a clear correlation between the monthly bill amount and the likelihood of cancellation (churn):

- Customers in the **Low** category are incredibly loyal, with a churn rate of only **11.2%.**
- The flight risk increases dramatically as the price goes up, hitting a critical peak of **37.5%** for the **High** category.
- Interestingly, we see a slight drop in churn (**32.9%**) for the **Premium** category (the most expensive bills).

3.5. Does `TotalCharges` affects `Churn` ?

The `TotalCharges` column is the massive, cumulative sum of all the money that the customer has ever paid the company since their very first day of service.

So it is related to both `tenure` and `MonthlyCharges` and we expect it to be a strong predictor of `Churn`.

🔴 We should convert the 'TotalCharges' column to a numeric type first. ⬇️

In [313…
```
telco['TotalCharges'] = pd.to_numeric(telco['TotalCharges'], errors='coerce')
```

Now let's see the distribution of the data ⬇️

In [314…
```
print(telco['TotalCharges'].describe())
print(f"\n0️⃣ NaN values : {telco['TotalCharges'].isna().sum()}")
```

```
count    7032.000000
mean     2283.300441
std      2266.771362
min        18.800000
25%       401.450000
50%      1397.475000
75%      3794.737500
max      8684.800000
Name: TotalCharges, dtype: float64
```

0️⃣ NaN values : 11

Let's redo the **Quantile Binning** for this column also

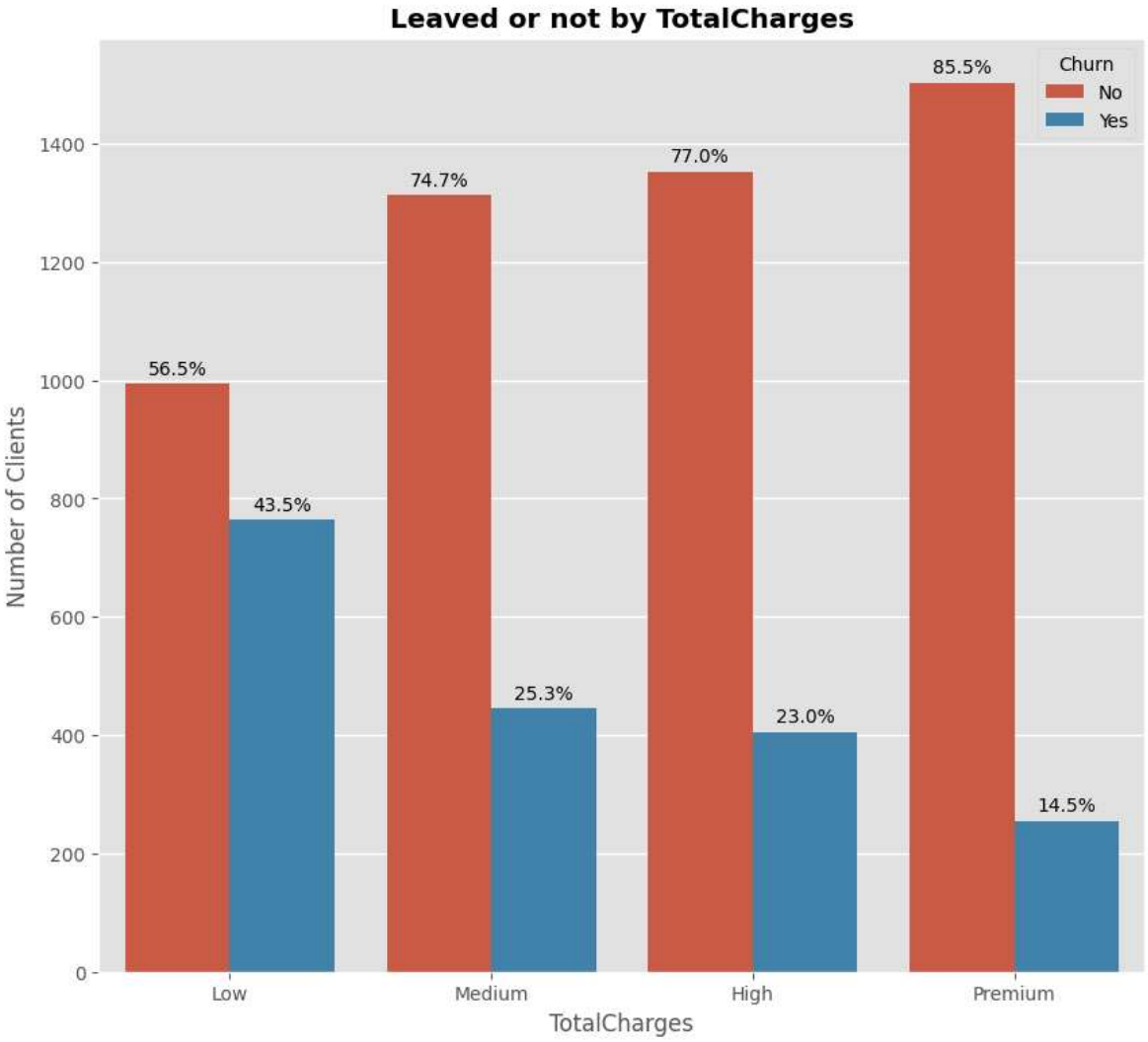
The Tiers: Using the dataset's statistical percentiles, we will classify the total charges into the following categories:

Price Category	Percentile Range	Charge Threshold
Low	0% - 25%	<= $401.45
Medium	25% - 50%	401.46—1397.48
High	50% - 75%	1397.49—3794.74
Premium	75% - 100%	> $3794.75

In [315…
```
telco['TotalCharges_categorial'] = pd.qcut(telco['TotalCharges'],
                                   q=4,
                                   labels=['Low', 'Medium', 'High', 'Premium'],
                                   retbins=False,
                                   precision=3,
                                   duplicates='raise')

telco['TotalCharges_categorial'].head()
```

Out[315…
```
0     Low
1    High
2     Low
3    High
4     Low
Name: TotalCharges_categorial, dtype: category
Categories (4, str): ['Low' < 'Medium' < 'High' < 'Premium']
```

In [316…
```
plt.figure(figsize=(10, 9))

ax = sns.countplot(telco, x="TotalCharges_categorial", hue="Churn")
plt.title('Leaved or not by TotalCharges', fontweight='bold')
plt.xlabel('TotalCharges')
plt.ylabel('Number of Clients')

tcharges_totals = telco['TotalCharges_categorial'].value_counts()

for c in ax.containers:
    labels = []
    for i, bar in enumerate(c):
        tcharges_name = ax.get_xticklabels()[i].get_text()
        percentage = (bar.get_height() / tcharges_totals[tcharges_name]) * 100
        labels.append(f'{percentage:.1f}%')
    ax.bar_label(c, labels=labels, padding=3)

plt.show()
```



📊 Insight

Observation: This chart reveals a fascinating reverse trend compared to our Monthly Charges analysis. As a customer's `TotalCharges` category increases, their likelihood of churning drastically *decreases*:

- Customers in the **Low** total charges tier have the highest flight risk by far at **43.5%**.
- As total charges accumulate over time, loyalty steadily increases, culminating in a highly secure **14.5%** churn rate for the **Premium** tier.

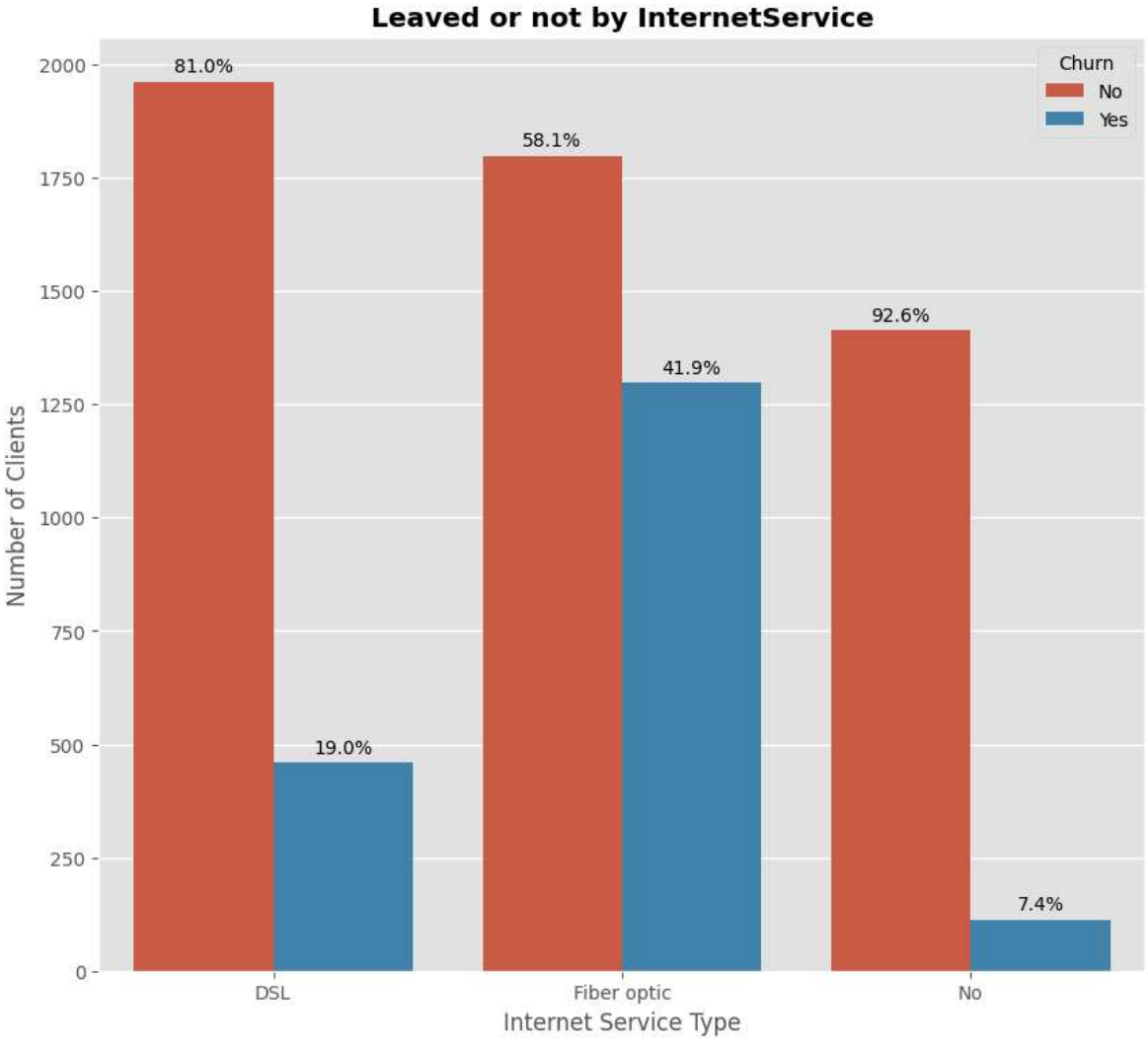
Business Context: At first glance, this looks like a paradox: *Why are the people who have paid the company the most money the least likely to leave?* The secret lies in its relationship with time. `TotalCharges` represents the customer's "Lifetime Value." To reach the "High" or "Premium" tier of total charges, a customer must have remained with the company for several years. As we established in our `tenure` analysis, customers who survive the initial onboarding phase become fiercely loyal. Conversely, the "Low" tier is primarily composed of brand-new, unestablished customers who are inherently at the highest risk of cancellation.

3.6. Does `InternetService` affects `Churn` ?

In [317…
```
telco['InternetService'].value_counts()
```

Out[317…
```
InternetService
Fiber optic    3096
DSL            2421
No             1526
Name: count, dtype: int64
```

In [318…
```
plt.figure(figsize=(10, 9))

ax = sns.countplot(telco, x="InternetService", hue="Churn")
plt.title('Leaved or not by InternetService', fontweight='bold')
plt.xlabel('Internet Service Type')
plt.ylabel('Number of Clients')

iServices_totals = telco['InternetService'].value_counts()

for c in ax.containers:
    labels = []
    for i, bar in enumerate(c):
        iServices_name = ax.get_xticklabels()[i].get_text()
        percentage = (bar.get_height() / iServices_totals[iServices_name]) * 100
        labels.append(f'{percentage:.1f}%')
    ax.bar_label(c, labels=labels, padding=3)

plt.show()
```



📊 Insight

Observation: This visualization uncovers a surprising and critical trend regarding the company's internet service types:

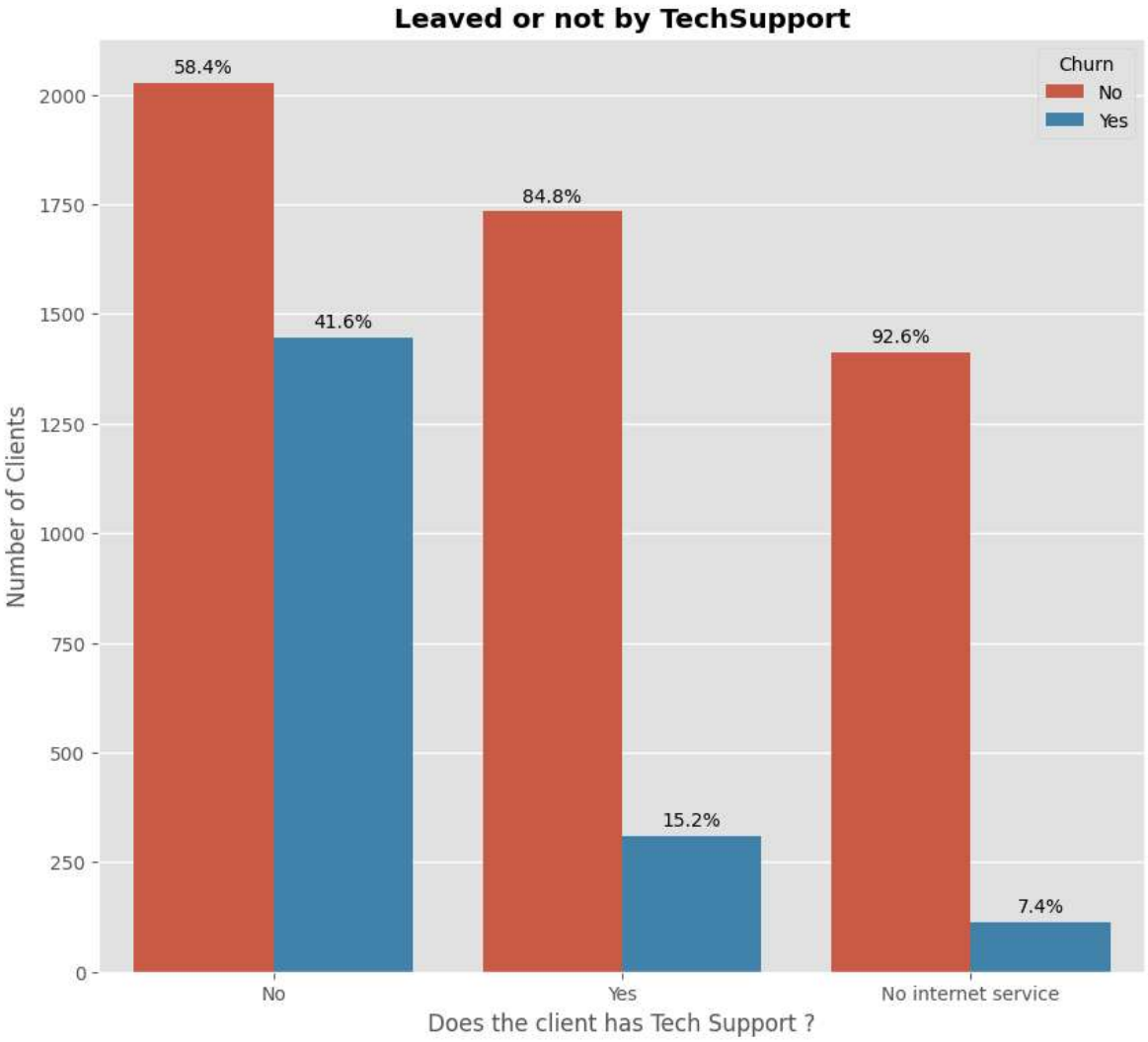
- Customers with **Fiber optic** connections have an exceptionally high churn rate of **41.9%**.
- Customers with standard **DSL** connections are much more stable, with a churn rate of only **19.0%.**
- Customers with **No internet service** (likely basic landline users) are the most loyal, with a tiny churn rate of **7.4%.**

3.7. Does `TechSupport` affects `Churn` ?

In [319…
```
telco['TechSupport'].value_counts()
```

Out[319…
```
TechSupport
No                     3473
Yes                    2044
No internet service    1526
Name: count, dtype: int64
```

In [320…
```
plt.figure(figsize=(10, 9))

ax = sns.countplot(telco, x="TechSupport", hue="Churn")
plt.title('Leaved or not by TechSupport', fontweight='bold')
plt.xlabel('Does the client has Tech Support ?')
plt.ylabel('Number of Clients')

tSupport_totals = telco['TechSupport'].value_counts()

for c in ax.containers:
    labels = []
    for i, bar in enumerate(c):
        tSupport_name = ax.get_xticklabels()[i].get_text()
        percentage = (bar.get_height() / tSupport_totals[tSupport_name]) * 100
        labels.append(f'{percentage:.1f}%')
    ax.bar_label(c, labels=labels, padding=3)

plt.show()
```



📊 Insight

This chart highlights a massive gap in customer retention based on whether or not they have technical support:

- Customers **without** Tech Support have a severely high churn rate of **41.6%.**
- Customers **with** Tech Support are highly secure, with their churn rate dropping all the way down to **15.2%.**
- The "No internet service" group remains consistent at a baseline **7.4%** churn.

3.8. Which **services** affect more `Churn` ?

In [321…

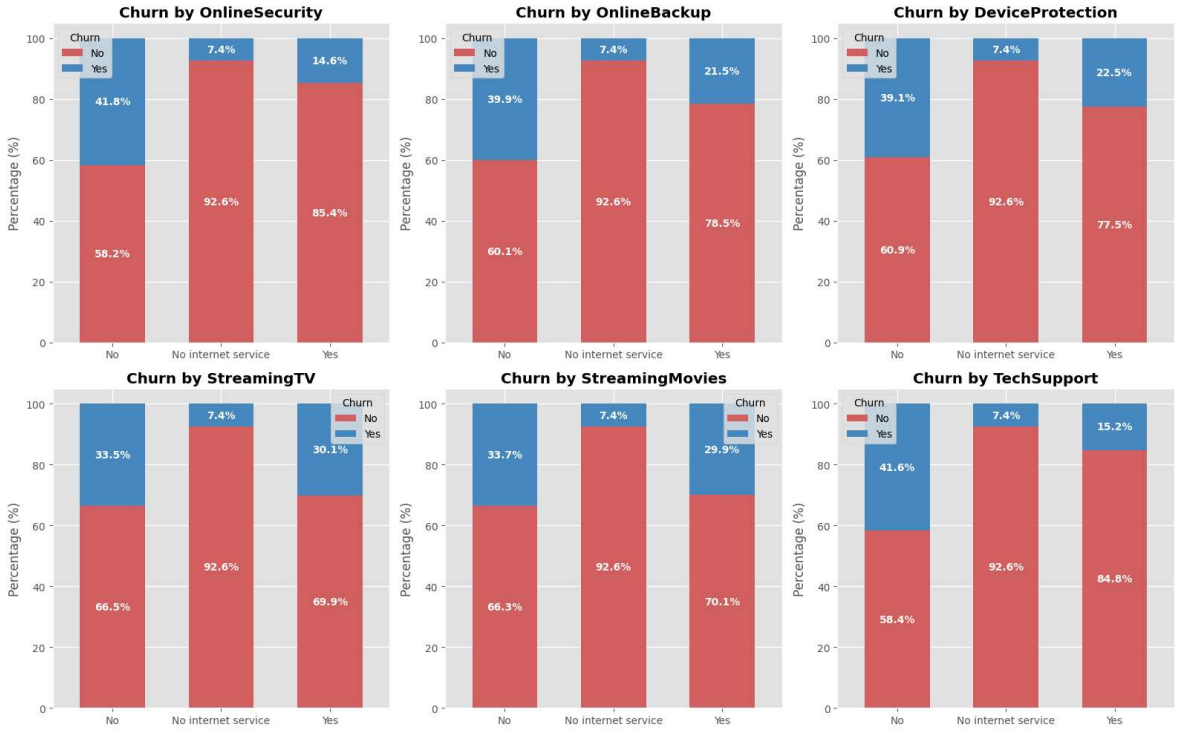
```
services = ['OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'StreamingTV',

summary_df = pd.DataFrame({s: telco[s].value_counts() for s in services})
print(summary_df)
```

	OnlineSecurity	OnlineBackup	DeviceProtection	\
No	3498	3088	3095	
Yes	2019	2429	2422	
No internet service	1526	1526	1526	

	StreamingTV	StreamingMovies	TechSupport
No	2810	2785	3473
Yes	2707	2732	2044
No internet service	1526	1526	1526

In [322…

```
# Create a figure and a 2x3 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(16, 10))

axes = axes.flatten()

for i, service in enumerate(services):
    cross_tab = pd.crosstab(telco[service], telco['Churn'], normalize='index') *

    # Plot directly onto the specific subplot axis
    cross_tab.plot(kind='bar', stacked=True, ax=axes[i], color=['#d65f5f', '#488

    axes[i].set_title(f'Churn by {service}', fontweight='bold')
    axes[i].set_xlabel('')
    axes[i].set_ylabel('Percentage (%)')
    axes[i].tick_params(axis='x', rotation=0)

    # Add the percentage text perfectly centered inside the bars
    for c in axes[i].containers:
        axes[i].bar_label(c, fmt='%.1f%%', label_type='center', color='white', f

plt.tight_layout() # So that titles don't overlap
plt.show()
```



📊 Insight: The Critical Importance of Security Services

Observation: This grid reveals a very clear divide between the different add-on services:

- **The Churn Shields:** Lacking `OnlineSecurity`, `TechSupport`, `OnlineBackup`, and `DeviceProtection` leads to massive churn rates (around 40%). Conversely, customers who have these services are highly loyal (churn drops drastically to 14.6% for Online Security and 15.2% for Tech Support).
- **The Streaming Illusion:** Surprisingly, `StreamingTV` and `StreamingMovies` have almost no impact on loyalty. Whether the customer has these options or not, the flight risk remains virtually identical (around 30-33%).

3.9. Do `PaperlessBilling` and `PaymentMethod` affect `Churn` ?

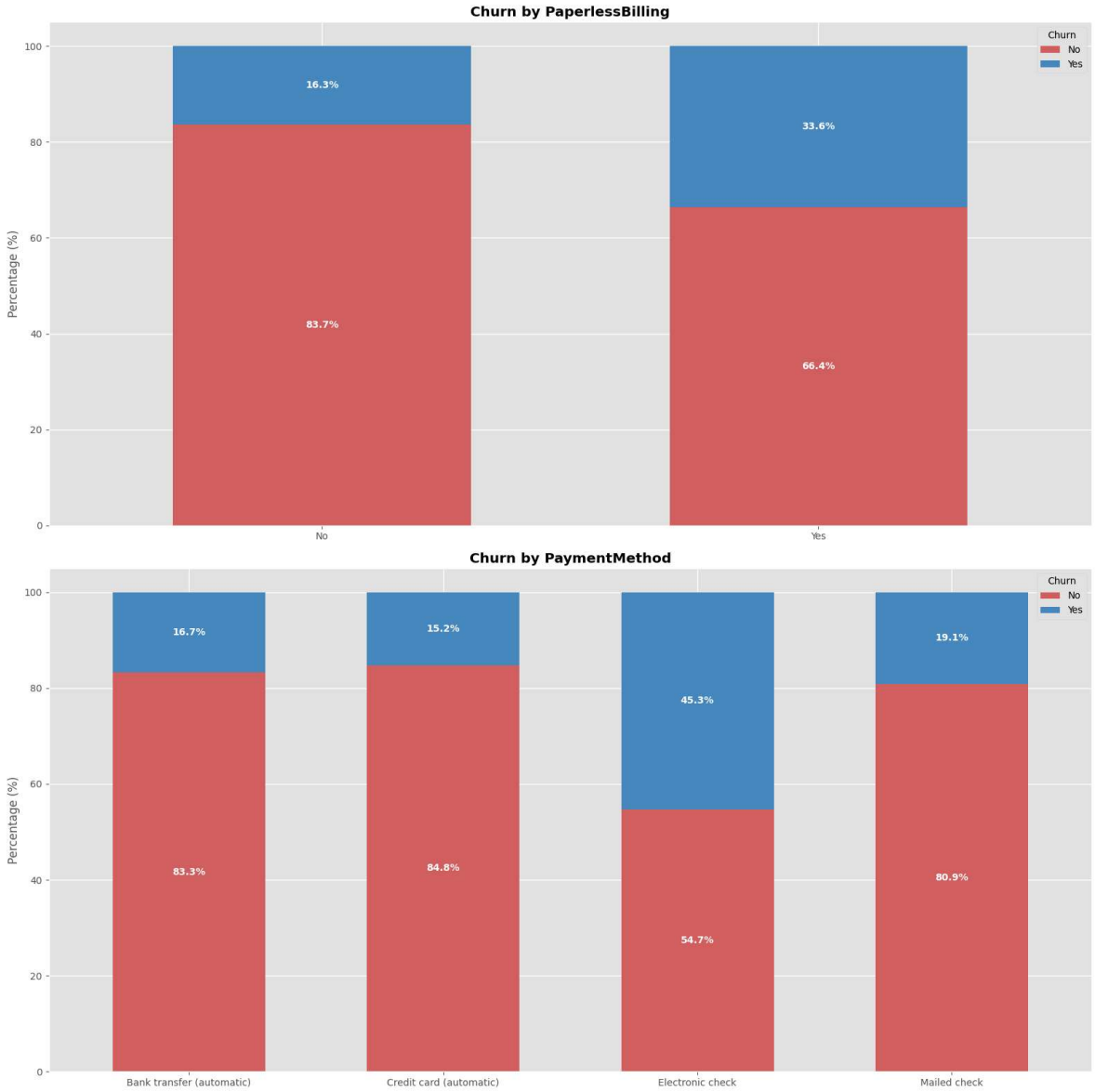
In [323…

```
payments = ['PaperlessBilling', 'PaymentMethod']

summary_df = pd.DataFrame({s: telco[s].value_counts() for s in payments})
print(summary_df)
```

```
                           PaperlessBilling  PaymentMethod
Bank transfer (automatic)               NaN         1544.0
Credit card (automatic)                 NaN         1522.0
Electronic check                        NaN         2365.0
Mailed check                            NaN         1612.0
No                                   2872.0            NaN
Yes                                  4171.0            NaN
```

In [324…

```
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(16, 16))

axes = axes.flatten()

for i, payment in enumerate(payments):
    cross_tab = pd.crosstab(telco[payment], telco['Churn'], normalize='index') *
    cross_tab.plot(kind='bar', stacked=True, ax=axes[i], color=['#d65f5f', '#488

    axes[i].set_title(f'Churn by {payment}', fontweight='bold')
    axes[i].set_xlabel('')
    axes[i].set_ylabel('Percentage (%)')
    axes[i].tick_params(axis='x', rotation=0)

    for c in axes[i].containers:
        axes[i].bar_label(c, fmt='%.1f%%', label_type='center', color='white', f

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



📊 Insight

Observation: These charts reveal significant disparities in churn based on how customers interact with their bills:

- **Paperless Billing:** Counterintuitively, customers who use paperless billing churn at more than double the rate (**33.6%**) compared to those who receive physical paper bills (**16.3%**).
- **Payment Method:** There is a massive anomaly with the **Electronic check** payment method, which drives a severe churn rate of **45.3%**. Meanwhile, automatic methods (Credit card and Bank transfer) are highly secure, with churn sitting comfortably around **15% to 16%.**

3.10. Which are the other columns that affect `Churn` ?

In [325…

```
others = ['SeniorCitizen', 'Partner', 'Dependents', 'PhoneService', 'MultipleLin

summary_df = pd.DataFrame({s: telco[s].astype(str).value_counts() for s in other
print(summary_df)
```

```
                  SeniorCitizen  Partner  Dependents  PhoneService  \
0                        5901.0      NaN         NaN           NaN
1                        1142.0      NaN         NaN           NaN
No                          NaN   3641.0      4933.0         682.0
No phone service            NaN      NaN         NaN           NaN
Yes                         NaN   3402.0      2110.0        6361.0

                  MultipleLines
0                           NaN
1                           NaN
No                       3390.0
No phone service          682.0
Yes                      2971.0
```

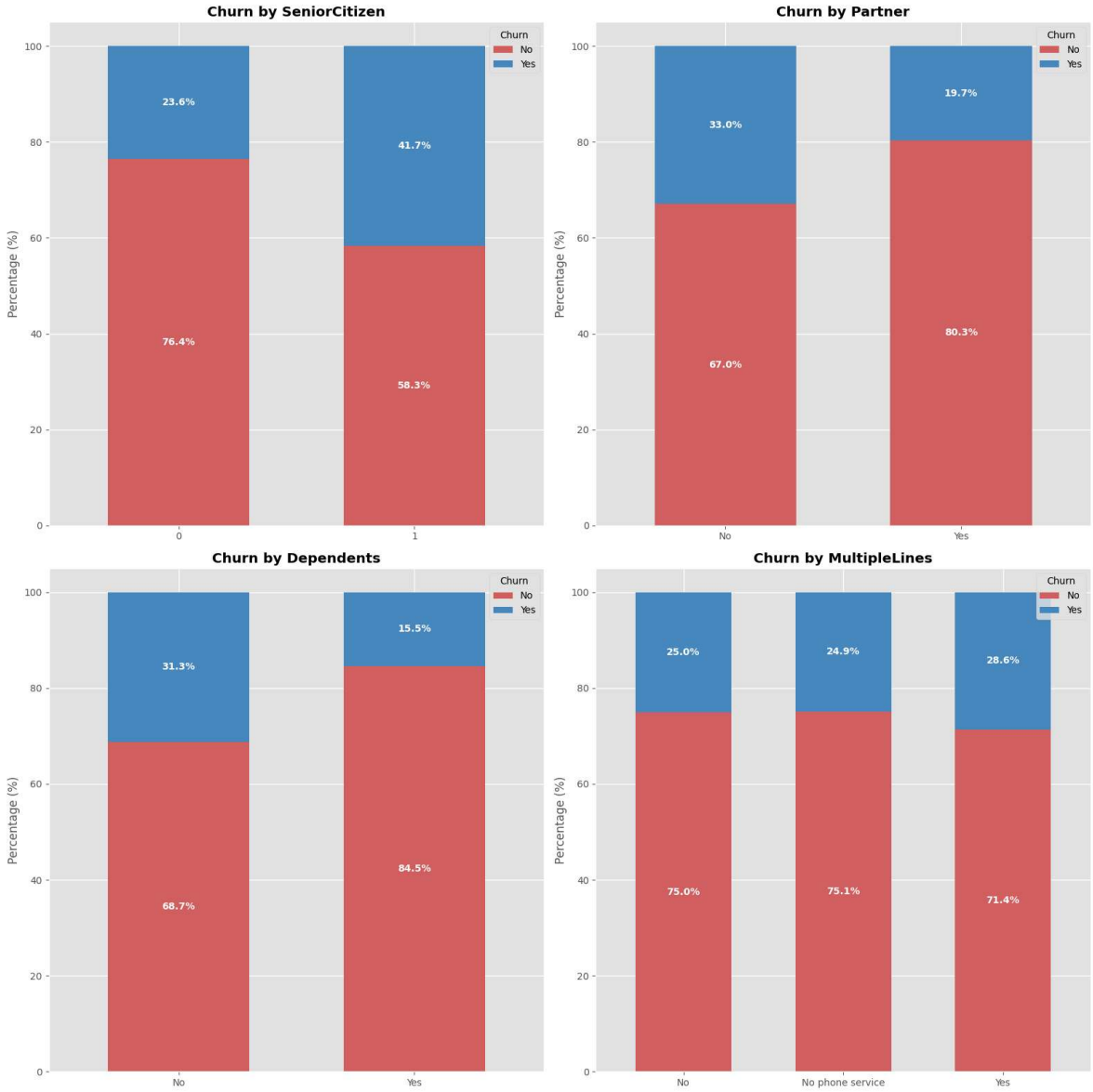
By comparing our dataset's value counts, we uncovered a perfect data overlap between two columns:

- Exactly **682** customers answered `"No"` in the `PhoneService` column.
- Exactly **682** customers are categorized as `"No phone service"` in the `MultipleLines` column.

That's why we can safely drop the `PhoneService` column from our dataset. We can safely drop this column because it introduces a mathematical issue called **multicollinearity**.

In [326…

```
others = ['SeniorCitizen', 'Partner', 'Dependents', 'MultipleLines']

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(16, 16))

axes = axes.flatten()

for i, other in enumerate(others):
    cross_tab = pd.crosstab(telco[other], telco['Churn'], normalize='index') * 1
    cross_tab.plot(kind='bar', stacked=True, ax=axes[i], color=['#d65f5f', '#488

    axes[i].set_title(f'Churn by {other}', fontweight='bold')
    axes[i].set_xlabel('')
    axes[i].set_ylabel('Percentage (%)')
    axes[i].tick_params(axis='x', rotation=0)

    for c in axes[i].containers:
        axes[i].bar_label(c, fmt='%.1f%%', label_type='center', color='white', f

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



📊 Insight

Observation: These four charts finalize our demographic and account analysis, revealing a very clear picture of who is leaving:

- **Family Ties:** Customers with a **Partner** (19.7% churn) and **Dependents** (15.5% churn) are highly loyal. Conversely, single customers and those without children churn at drastically higher rates (33.0% and 31.3%, respectively).
- **Senior Risk: Senior Citizens** have an alarmingly high churn rate of **41.7%**, which is nearly double the rate of non-seniors (23.6%).
- **Multiple Lines is Noise:** Whether a customer has **MultipleLines** or a single line barely moves the needle. The churn rate hovers consistently between 24.9% and 28.6% across all categories.

4. 🧹 Data Cleaning & Imputation

4.1. Cleaning NaN values

In [327…
```
print(telco.isna().sum())
```

```
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
MonthlyCharges_categorial   0
TotalCharges_categorial    11
dtype: int64
```

Perfect ! We don't have missing values.

We can drop the 11 lines with missing values in MonthlyCharges. It is les than 1% of the data.

In [328…
```
telco.dropna(subset=['TotalCharges'], inplace=True)
print(telco.isna().sum())
```

```
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
MonthlyCharges_categorial   0
TotalCharges_categorial     0
dtype: int64
```

4.2. Dropping useless columns

In [329…
```
columns_to_drop = ['customerID','MultipleLines', 'PhoneService', 'gender', 'Mont
telco.drop(columns=columns_to_drop, inplace=True)
```

In [330…
```
print(f"New dataset shape: {telco.shape}")
print(f"New dataset columns: {telco.columns}")
```

```
New dataset shape: (7032, 15)
New dataset columns: Index(['SeniorCitizen', 'Partner', 'Dependents', 'tenure',
'InternetService',
       'OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupport',
       'Contract', 'PaperlessBilling', 'PaymentMethod', 'MonthlyCharges',
       'TotalCharges', 'Churn'],
      dtype='str')
```

5. 🧬 Feature Engineering

5.1. Creating a `SecurityScore` column

This column will regroup 4 columns : `OnlineSecurity` , `OnlineBackup` , `DeviceProtection` , `TechSupport` . As we've seen in the EDA, customers with protection services are much more loyal.

So the higher the value of this column, the more loyal the customer is.

In [331…
```
security_cols = ['OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupp
telco['SecurityScore'] = (telco[security_cols] == 'Yes').sum(axis=1)

# Check
telco['SecurityScore'].value_counts().sort_index()
```

Out[331…
```
SecurityScore
0    2787
1    1467
2    1372
3     937
4     469
Name: count, dtype: int64
```

In [332…
```
columns_to_drop = ['OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSu
telco.drop(columns=columns_to_drop, inplace=True)
```

5.2. Creating a `HasFamily` column

This column will regroup 4 columns : `Partner` , `Dependents` . As we've seen in the EDA, customers having family/friends are much more loyal.

So the higher the value of this column, the more loyal the customer is.

In [333…
```
family_cols = ['Partner', 'Dependents']
telco['HasFamily'] = (telco[family_cols] == 'Yes').sum(axis=1)
```

```
# Check
telco['HasFamily'].value_counts().sort_index()
```

Out[333…
```
HasFamily
0    3280
1    2012
2    1740
Name: count, dtype: int64
```

In [334…
```
columns_to_drop = ['Partner', 'Dependents']
telco.drop(columns=columns_to_drop, inplace=True)
```

5.3. Creating a `IsFiber` column

Because clients with a fiber `InternetService` = Fiber optic have a higher Churn rate

In [335…
```
telco['isFiber'] = (telco['InternetService'] == 'Fiber optic').astype(int)

# Check
telco['isFiber'].value_counts().sort_index()
```

Out[335…
```
isFiber
0    3936
1    3096
Name: count, dtype: int64
```

In [336…
```
columns_to_drop = ['InternetService']
telco.drop(columns=columns_to_drop, inplace=True)
```

In [337…
```
print(telco.isna().sum())
```

```
SeniorCitizen       0
tenure              0
Contract            0
PaperlessBilling    0
PaymentMethod       0
MonthlyCharges      0
TotalCharges        0
Churn               0
SecurityScore       0
HasFamily           0
isFiber             0
dtype: int64
```

6. 🔢 Preprocessing & Encoding

In [338…
```
telco.head()
```

Out[338…

	SeniorCitizen	tenure	Contract	PaperlessBilling	PaymentMethod	MonthlyCharges
0	0	1	Month-to-month	Yes	Electronic check	29.85
1	0	34	One year	No	Mailed check	56.95
2	0	2	Month-to-month	Yes	Mailed check	53.85
3	0	45	One year	No	Bank transfer (automatic)	42.30
4	0	2	Month-to-month	Yes	Electronic check	70.70

6.1. PaperlessBilling Column

In [339…

```
# PaperlessBilling Column
print(telco['PaperlessBilling'].value_counts())

telco['PaperlessBilling'] = telco['PaperlessBilling'].map({"Yes": 1, "No": 0})

print(telco['PaperlessBilling'].value_counts())
```

```
PaperlessBilling
Yes    4168
No     2864
Name: count, dtype: int64
PaperlessBilling
1    4168
0    2864
Name: count, dtype: int64
```

6.2. Churn Column

In [340…

```
# Churn Column
print(telco['Churn'].value_counts())

telco['Churn'] = telco['Churn'].map({"Yes": 1, "No": 0})

print(telco['Churn'].value_counts())
```

```
Churn
No     5163
Yes    1869
Name: count, dtype: int64
Churn
0    5163
1    1869
Name: count, dtype: int64
```

6.3. PaymentMethod Column

In [341…
```
#  PaymentMethod Column
print(telco['PaymentMethod'].value_counts())

telco['PM_Electronic_check'] = (telco['PaymentMethod'] == 'Electronic check').as
telco['PM_Mailed_check'] = (telco['PaymentMethod'] == 'Mailed check').astype(int
telco['PM_Bank_transfer_automatic'] = (telco['PaymentMethod'] == 'Bank transfer
telco['PM_Credit_card_automatic'] = (telco['PaymentMethod'] == 'Credit card (aut

print(telco['PM_Electronic_check'].value_counts())
print(telco['PM_Mailed_check'].value_counts())
print(telco['PM_Bank_transfer_automatic'].value_counts())
print(telco['PM_Credit_card_automatic'].value_counts())

telco.drop(columns=['PaymentMethod'], inplace=True)
```

```
PaymentMethod
Electronic check             2365
Mailed check                 1604
Bank transfer (automatic)    1542
Credit card (automatic)      1521
Name: count, dtype: int64
PM_Electronic_check
0    4667
1    2365
Name: count, dtype: int64
PM_Mailed_check
0    5428
1    1604
Name: count, dtype: int64
PM_Bank_transfer_automatic
0    5490
1    1542
Name: count, dtype: int64
PM_Credit_card_automatic
0    5511
1    1521
Name: count, dtype: int64
```

6.4. tenure Column

In [342…
```
# tenure column
def setTenure(value) :
    if value < 12 :
        return 0
    elif value < 24 :
        return 1
    else :
        return 2

telco['tenure'] = telco['tenure'].apply(setTenure)

print(telco['tenure'].value_counts())
```

```
tenure
2    3927
0    2058
1    1047
Name: count, dtype: int64
```

6.5. Contract Column

In [343…
```
# Contract column
print(telco['Contract'].value_counts())

telco['Contract'] = telco['Contract'].map({"Month-to-month": 1, "One year": 2, "

print(telco['Contract'].value_counts())
```

```
Contract
Month-to-month    3875
Two year          1685
One year          1472
Name: count, dtype: int64
Contract
1    3875
3    1685
2    1472
Name: count, dtype: int64
```

6.6. MonthlyCharges and TotalCharges Column

In [344…
```
# MonthlyCharges and TotalCharges column
telco['TotalCharges'] = pd.to_numeric(telco['TotalCharges'], errors='coerce').fi

scaler = RobustScaler()
cols_to_scale = ['MonthlyCharges', 'TotalCharges']
telco[cols_to_scale] = scaler.fit_transform(telco[cols_to_scale])
```

6.7. Quick Check

In [345…
```
telco.head()
```

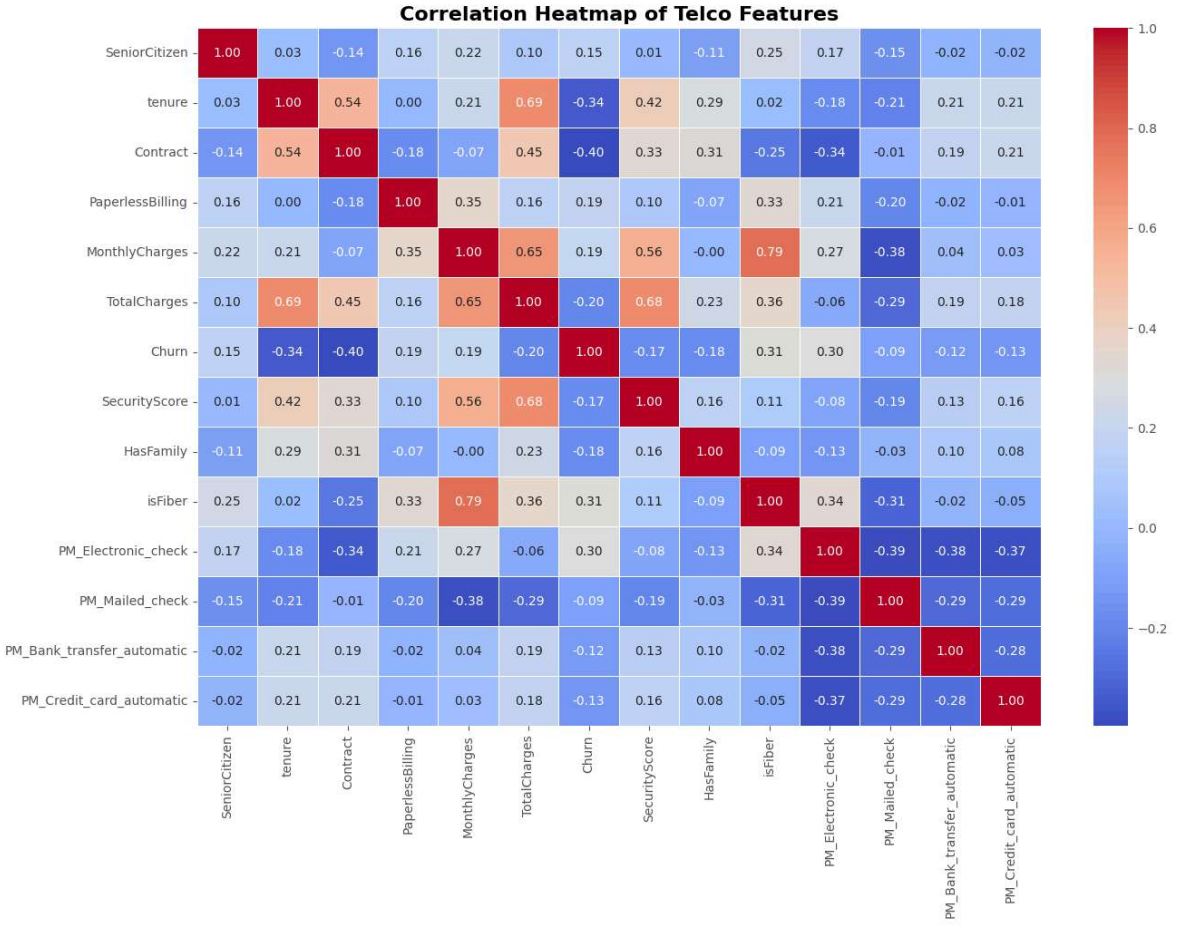
Out[345…

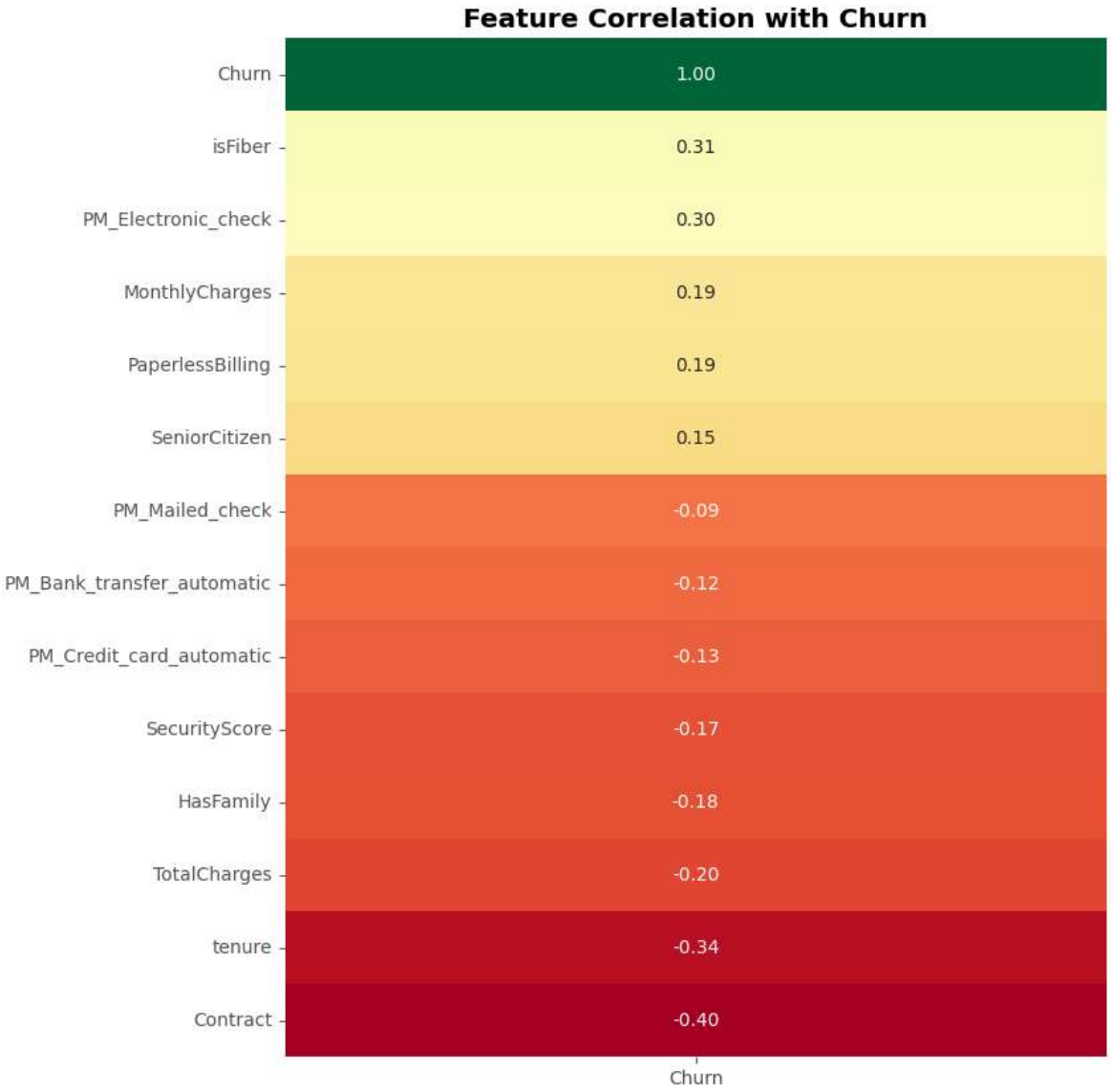
	SeniorCitizen	tenure	Contract	PaperlessBilling	MonthlyCharges	TotalCharges	Chu
0	0	0	1	1	-0.746200	-0.403038	
1	0	2	2	0	-0.246891	0.145000	
2	0	0	1	1	-0.304007	-0.379963	
3	0	2	2	0	-0.516813	0.130633	
4	0	0	1	1	0.006449	-0.367144	

6.8. Heatmap

In [346…
```
corr_matrix = telco.corr()

plt.figure(figsize=(15, 10))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm', linewidths=0.5)

plt.title('Correlation Heatmap of Telco Features', fontweight='bold', fontsize=1
plt.show()
```

```
plt.figure(figsize=(8, 10))
churn_corr = telco.corr()['Churn'].sort_values(ascending=False)
sns.heatmap(churn_corr.to_frame(), annot=True, fmt=".2f", cmap='RdYlGn', cbar=Fa
plt.title('Feature Correlation with Churn', fontweight='bold')

plt.show()
```





7. 🤖 Logistic Regression Model Building

7.1. Why start with Logistic Regression?

Before training our model, it is important to understand *why* we are choosing **Logistic Regression** as our first algorithm. Despite the word "regression" in its name, this is actually a powerful **classification** algorithm.

Here is why it is the industry standard for starting a Churn Prediction project:

- **1. It is designed for Binary Problems:** Our goal is to predict a simple Yes or No (1 or 0) outcome: *Will the customer leave?* Logistic Regression uses a mathematical curve (the Sigmoid function) to calculate the exact probability of an event happening, making it a perfect fit for this task.
- **2. Ultimate Interpretability:** In business, predicting *who* will leave is only half the battle; stakeholders want to know *why*. Unlike complex "black-box" algorithms (like Neural Networks or Random Forests), Logistic Regression provides clear

coefficients. It allows us to say exactly how much a specific feature increases or decreases the chance of churn.

- **3. The Perfect Baseline:** It is computationally fast, highly efficient, and resistant to minor noise. By establishing a solid baseline score with Logistic Regression, we have a clear mathematical benchmark. Later, if we want to test more complex algorithms, we can measure exactly how much they improve upon this baseline.

7.2. Defining Variables

In [347…
```
X = telco.drop(columns=['Churn'])
y = telco['Churn']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

print(f"Training set dimensions: {X_train.shape}")
print(f"Testing set dimensions: {X_test.shape}\n")
```

```
Training set dimensions: (5625, 13)
Testing set dimensions: (1407, 13)
```

7.3. Training and Testing

In [348…
```
log_model = LogisticRegression(max_iter=1000)
log_model.fit(X_train, y_train)

y_pred = log_model.predict(X_test)
```

7.4. Evaluating the Performance

In [349…
```
print(f"Overall Accuracy: {accuracy_score(y_test, y_pred):.2%}\n")
print("Detailed Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Overall Accuracy: 79.03%

Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.90      0.86      1033
           1       0.63      0.50      0.56       374

    accuracy                           0.79      1407
   macro avg       0.73      0.70      0.71      1407
weighted avg       0.78      0.79      0.78      1407
```

At first glance, an **Overall Accuracy of 79.03%** looks like a massive success. However, in Data Science—especially in churn prediction—overall accuracy can be highly misleading. We are dealing with an **imbalanced dataset** (1033 customers stayed, while only 374 left in our test set). Because the majority of customers stay, the model naturally biases its guesses toward "Stayed."

Classification Report:

✅ ***The Good***: Identifying Loyal Customers (Class 0)

Our model is incredibly good at recognizing customers who are not going to leave.
Recall (0.90):

⚠️ ***The Catch***: Missing the Churners (Class 1)

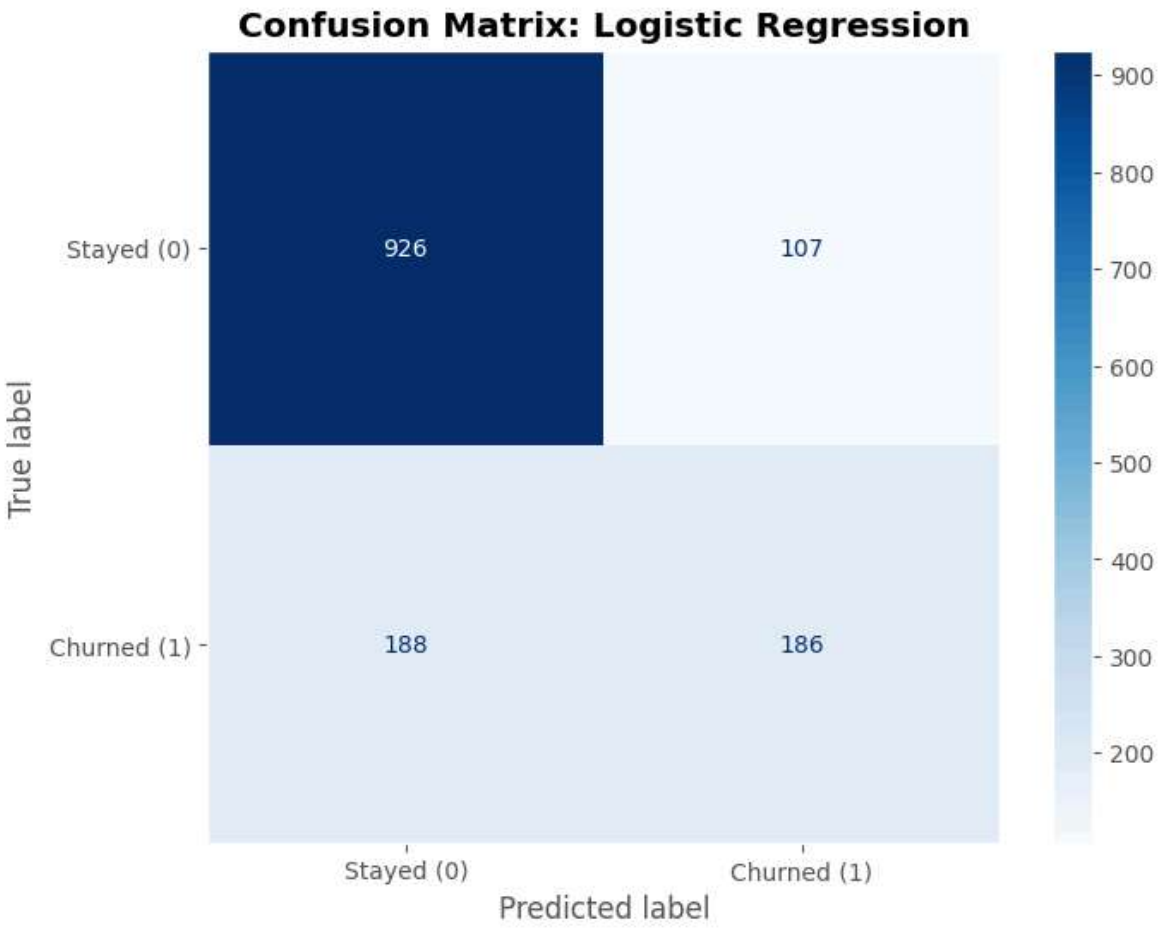
The `1` row represents the people who actually canceled their service. This is where the business loses money, and our model is currently struggling.

- **Recall (0.50):** This is the most critical metric in our entire project. It means that out of all the customers who actually left, our model only caught **50%** of them. The other half slipped through the cracks completely undetected (these are our **False Negatives**).
- **Precision (0.63):** When the model *does* raise the alarm and predict a customer will leave, it is only correct **63%** of the time.

7.5. Confusion Matrix

In [350…

```
fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay.from_predictions(
    y_test,
    y_pred,
    cmap='Blues',
    ax=ax,
    display_labels=['Stayed (0)', 'Churned (1)']
)

plt.title('Confusion Matrix: Logistic Regression', fontweight='bold')
plt.grid(False)
plt.show()
```

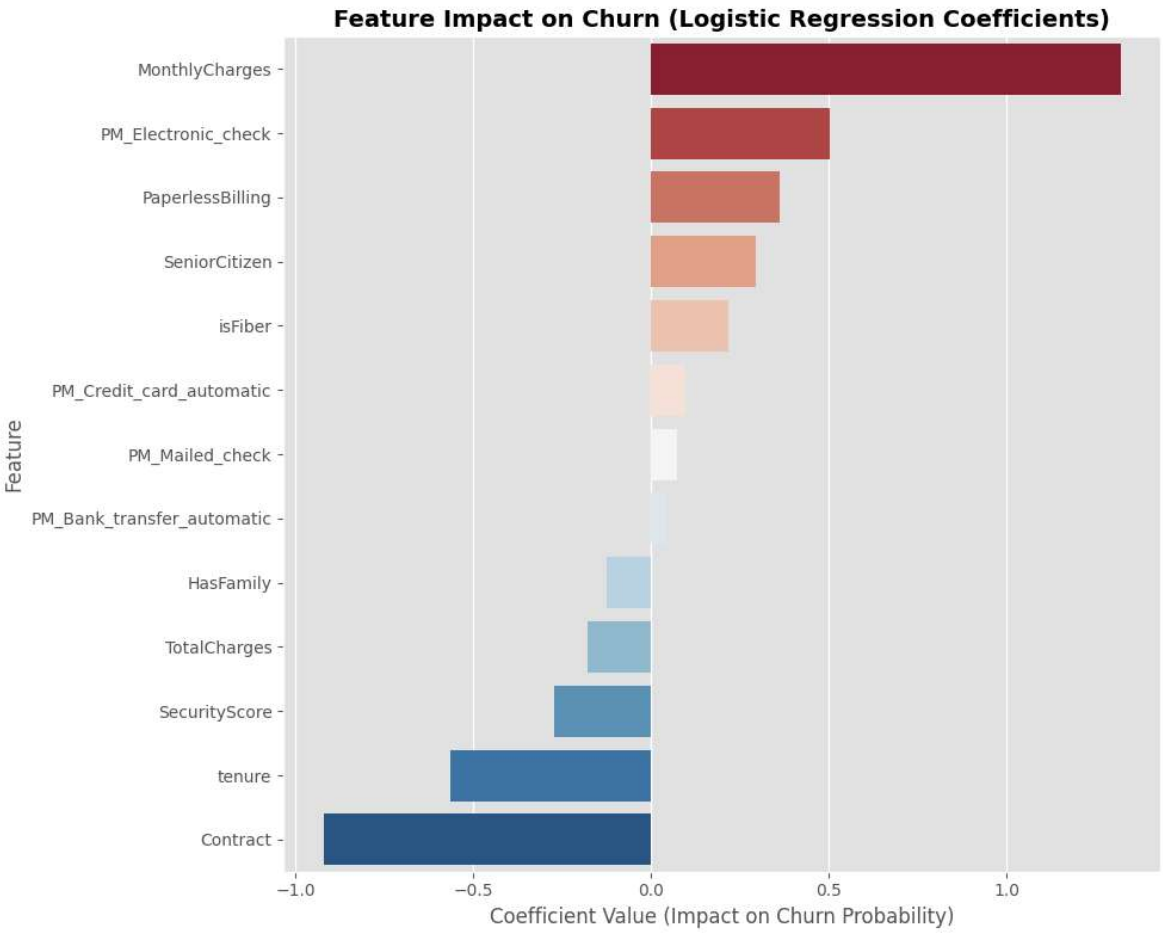


7.6. Model's Coefficients

Before we try to improve this Recall score (which we can do later by balancing the dataset or using a more complex algorithm like a Random Forest), we first need to extract the model's **Coefficients** to understand exactly *why* it is making these decisions.

In [351…

```
coefficients = log_model.coef_[0]

feature_names = X.columns
coef_df = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})

coef_df = coef_df.sort_values(by='Coefficient', ascending=False)

plt.figure(figsize=(10, 8))
sns.barplot(data=coef_df, x='Coefficient', y='Feature', palette='RdBu', hue='Fea
plt.title('Feature Impact on Churn (Logistic Regression Coefficients)', fontweig
plt.xlabel('Coefficient Value (Impact on Churn Probability)', fontsize=12)
plt.ylabel('Feature', fontsize=12)
plt.tight_layout()
plt.show()
```



🚨 Risk Factors : The features pulling to the right (positive coefficients) actively increase the probability of a customer leaving.

- **Monthly Charges**
- **Electronic Checks**
- **Fiber Optic & Paperless Billing**

🛡️ Loyalty Anchors : The features pulling to the left (negative coefficients) actively protect the company from churn.

- **Contract Length**
- **Tenure**
- **Security Score & Family**

7.7. Resampling with SMOTE

As we saw in our baseline evaluation, our model suffers from a low **Recall (50%)** for the minority class. It simply hasn't seen enough examples of customers leaving to confidently recognize the complex patterns behind churn.

To fix this, we are going to use a powerful Data Science technique called **SMOTE** (Synthetic Minority Over-sampling Technique).

- 🧬 **What is SMOTE?** Instead of simply duplicating our existing churn data (which can lead to overfitting), SMOTE uses a Nearest Neighbors algorithm to generate mathematically realistic "synthetic" data points. It plots our actual churners in a

multi-dimensional space, draws lines between them, and creates brand-new, artificial customers along those lines.

- 🎯 **Why are we using it?** By applying SMOTE to our training data, we will force the minority class (Churners) to be exactly equal in size to the majority class (Loyal Customers).

- **The Goal:** By providing the model with a perfectly balanced dataset to learn from, we aim to significantly *boost our Recall* score and finally catch the churners who were slipping through the cracks.

(Note: We strictly apply SMOTE to the Training set only. The Test set must remain untouched to evaluate the model on real-world, imbalanced data).

In [352…

```
smote = SMOTE(random_state=42) # This will create synthetic examples of the mino


# Apply SMOTE to the TRAINING data only
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

print(f"Original training shape: {X_train.shape}")
print(f"Resampled training shape: {X_train_resampled.shape}")
print(f"------------------------------------------------")

print(f"New Class Distribution:\n{y_train_resampled.value_counts()}\n")

log_model_smote = LogisticRegression(max_iter=1000)
log_model_smote.fit(X_train_resampled, y_train_resampled)
y_pred_smote = log_model_smote.predict(X_test)

print(f"------------------------------------------------")
print(f"Overall Accuracy after SMOTE: {accuracy_score(y_test, y_pred_smote):.2%}")
print("Detailed Classification Report (SMOTE):")
print(classification_report(y_test, y_pred_smote))
```

```
Original training shape: (5625, 13)
Resampled training shape: (8260, 13)
------------------------------------------------
New Class Distribution:
Churn
0    4130
1    4130
Name: count, dtype: int64

------------------------------------------------
Overall Accuracy after SMOTE: 73.13%
Detailed Classification Report (SMOTE):
              precision    recall  f1-score   support

           0       0.89      0.72      0.80      1033
           1       0.50      0.76      0.60       374

    accuracy                           0.73      1407
   macro avg       0.70      0.74      0.70      1407
weighted avg       0.79      0.73      0.75      1407
```

After applying **SMOTE** to balance our training data, we see a significant shift in the model's performance. We have successfully traded overall accuracy for a much higher "catch rate" of at-risk customers.

📊 Performance Comparison

Metric	Before SMOTE	After SMOTE
Overall Accuracy	79.03%	73.13%
Recall (Churners)	0.50	**0.76**
Precision (Churners)	0.63	0.50

🧠 Key Observations

- **The Recall Success:** Our model now identifies **76%** of customers who actually churn, compared to only 50% previously. This is a massive improvement for the business, as we are now missing far fewer at-risk individuals.
- **The Precision-Recall Trade-off:** To catch more churners, the model has become "more suspicious." While it catches more true churners, it also incorrectly flags more loyal customers (Precision dropped to 0.50).
- **The Business Verdict:** This model is significantly more valuable than the baseline. In a churn context, the cost of a **False Negative** (losing a customer you didn't see coming) is much higher than the cost of a **False Positive** (sending a discount to a loyal customer).

7.8. Optimizing the parameters using **GridSearchCV**

In [353…

```
param_grid = [
    {
        'penalty': ['l1', 'l2'],
        'C': [0.01, 0.1, 1, 10, 100],
        'solver': ['liblinear']
    },
    {
        'penalty': ['l2'],
        'C': [0.01, 0.1, 1, 10, 100],
        'solver': ['lbfgs']
    }
]

grid_search = GridSearchCV(estimator=LogisticRegression(max_iter=1000),
                           param_grid=param_grid,
                           cv=5,
                           scoring='recall',
                           verbose=0,
                           n_jobs=-1)

grid_search.fit(X_train_resampled, y_train_resampled)

print(f"Best Recall Score during CV: {grid_search.best_score_:.2%}")
print("Best Parameters:", grid_search.best_params_)
```

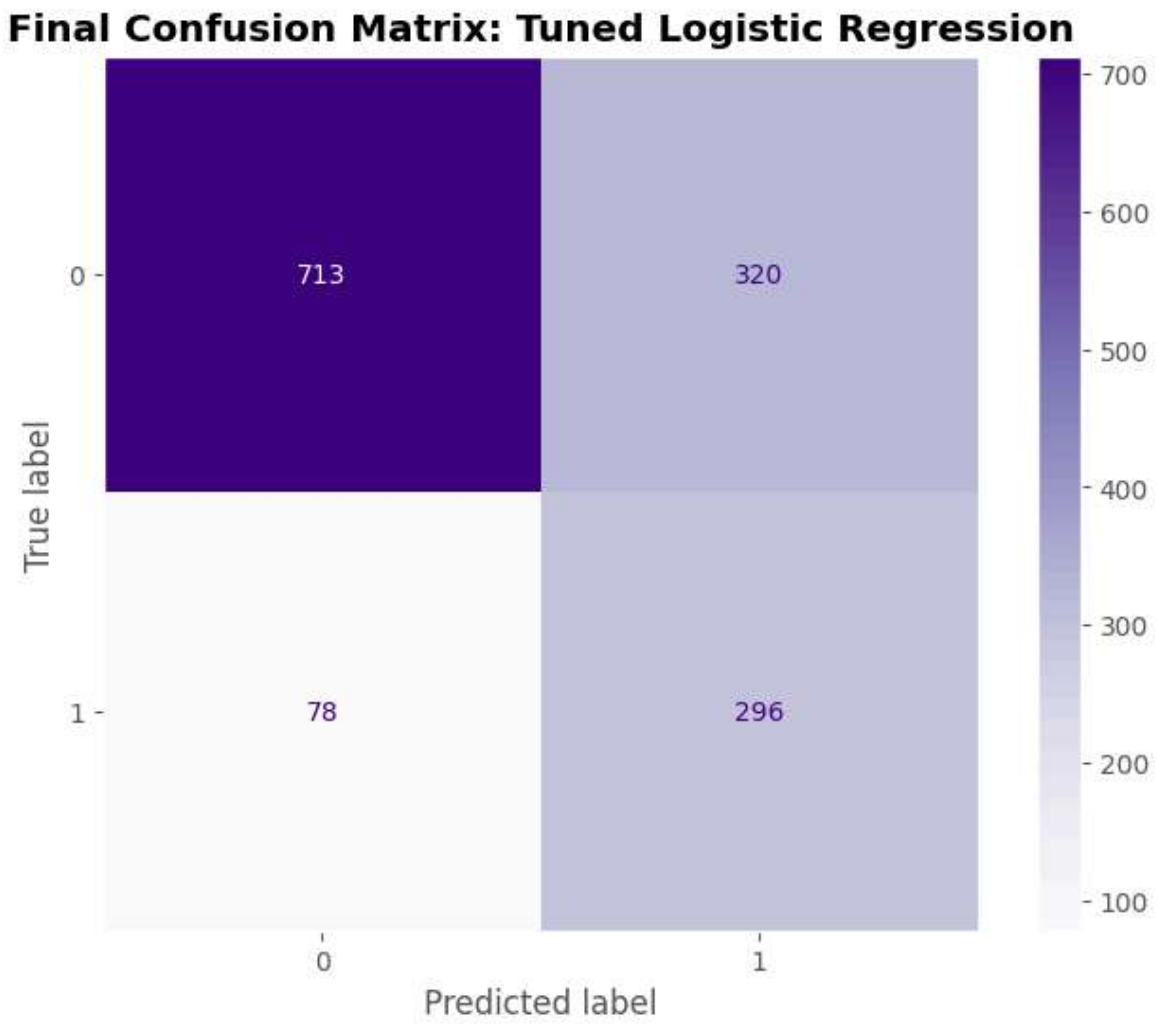
```
Best Recall Score during CV: 84.38%
Best Parameters: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
```

```
d:\Personal_projects\Telco-Model\.venv\Lib\site-packages\sklearn\linear_model\_logistic.py:1135: FutureWarning: 'penalty' was deprecated in version 1.8 and will be removed in 1.10. To avoid this warning, leave 'penalty' set to its default value and use 'l1_ratio' or 'C' instead. Use l1_ratio=0 instead of penalty='l2', l1_ratio=1 instead of penalty='l1', and C=np.inf instead of penalty=None.
  warnings.warn(
```

In [354…

```python
best_log_model = grid_search.best_estimator_

y_pred_final_log = best_log_model.predict(X_test)

print("--- Final Optimized Logistic Regression Results ---")
print(f"Overall Accuracy: {accuracy_score(y_test, y_pred_final_log):.2%}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_final_log))


fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_final_log, cmap='Purples'
plt.title('Final Confusion Matrix: Tuned Logistic Regression', fontweight='bold'
plt.grid(False)
plt.show()
```

```
--- Final Optimized Logistic Regression Results ---
Overall Accuracy: 71.71%

Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.69      0.78      1033
           1       0.48      0.79      0.60       374

    accuracy                           0.72      1407
   macro avg       0.69      0.74      0.69      1407
weighted avg       0.79      0.72      0.73      1407
```



7.9. Strategic Trade-offs: Understanding the "Shift" in Errors

After tuning our model with **SMOTE** and **GridSearchCV**, you may notice that while we are catching more churners, the total number of errors—specifically **False Positives (FP)** and **False Negatives (FN)**—has shifted. This is not a failure of the model, but a calculated business trade-off.

📊 The Evolution of our Confusion Matrix

- **Reduction in False Negatives (The Wins):** Our primary goal was to reduce the number of customers who leave without being detected. We successfully dropped the FN count significantly (catching **79%** of churners). These represent saved revenue.
- **Increase in False Positives (The Cost):** By making the model more sensitive ("suspicious"), we naturally increased the number of loyal customers flagged as churners.

⚖️ The Business ROI: Why this is better

In Churn Prediction, not all errors are equal. We have moved from a "balanced" error state to a **"Business-Optimal"** error state:

1. **Cost of a False Negative (Missing a Churner): HIGH.** You lose the entire lifetime value of that customer plus the high cost of acquiring a replacement.
2. **Cost of a False Positive (Flagging a Loyal User): LOW.** The cost is merely the price of a marketing email or a small discount offer.

🏁 Conclusion: Reaching the Linear Limit

Our **Tuned Logistic Regression** is now as efficient as a linear model can be. It has identified the "sweet spot" where it saves the most money for the company by prioritizing **Recall**.

However, because the model is linear, it can only separate classes with a straight mathematical boundary. To further reduce the **False Positives** without losing our **high Recall**, we must now transition to **Non-Linear Ensemble Models** like Random Forest.

8. 🤖 XGBOOST Model Building

Now that we have a solid baseline from Logistic Regression, we will implement **XGBoost**.

🚀 Why XGBoost?

- **Boosting Logic:** Unlike previous models that work in parallel, XGBoost builds trees one by one. Each new tree learns from the errors (residuals) of the previous one, making it incredibly precise.
- **Handling Non-Linearity:** It can capture complex "if-this-then-that" relationships that Logistic Regression completely misses.
- **Speed and Performance:** It is highly optimized for system resources and is famous for winning Kaggle competitions.

We will continue using our **SMOTE-balanced data** to ensure XGBoost stays focused on the churners.

8.1. XGBoost Original Data

In [355…

```
X = telco.drop(columns=['Churn'])
y = telco['Churn']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

xgb_model = XGBClassifier(
    n_estimators=100,
    max_depth=4,
    learning_rate=0.1,
    random_state=42,
    eval_metric='logloss'
)
xgb_model.fit(X_train, y_train)

y_pred_xgb = xgb_model.predict(X_test)

print(f"Overall Accuracy: {accuracy_score(y_test, y_pred_xgb):.2%}\n")
```

```
print("Detailed Classification Report:")
print(classification_report(y_test, y_pred_xgb))
```

```
Overall Accuracy: 79.10%

Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.88      0.86      1033
           1       0.62      0.54      0.58       374

    accuracy                           0.79      1407
   macro avg       0.73      0.71      0.72      1407
weighted avg       0.78      0.79      0.79      1407
```

📊 Performance vs. Baseline Logistic Regression

Metric	Baseline Logistic (Original)	XGBoost (Original)	Change
Overall Accuracy	79.03%	**79.10%**	📈 +0.07%
Recall (Churners)	0.50	**0.54**	🚀 **+0.04**
Precision (Churners)	0.63	**0.62**	📉 -0.01

🧠 Key Insights

- **Non-Linear Advantage:** Even without SMOTE, XGBoost managed to catch **4% more churners** than Logistic Regression. This indicates that the "Boosting" trees are successfully identifying complex, non-linear relationships between features (e.g., how `MonthlyCharges` interacts with `tenure` and `InternetService`) that a linear model misses.
- **Stable Precision:** The model maintained a solid Precision of 62%, meaning it is still fairly cautious and accurate when it predicts a customer will leave.
- **The Complexity Gap:** While 54% Recall is an improvement, it is still not enough for a proactive business strategy. This confirms that the imbalance in the data is a significant hurdle that even advanced algorithms struggle to overcome on their own.

8.2. XGBoost + SMOTE

In [356…

```
xgb_smote = XGBClassifier(
    n_estimators=100,
    max_depth=4,
    learning_rate=0.1,
    random_state=42,
    eval_metric='logloss'
)
xgb_smote.fit(X_train_resampled, y_train_resampled)

y_pred_xgb_smote = xgb_smote.predict(X_test)

print(f"Overall Accuracy: {accuracy_score(y_test, y_pred_xgb_smote):.2%}\n")
print("Detailed Classification Report:")
print(classification_report(y_test, y_pred_xgb_smote))
```

```
Overall Accuracy: 73.49%

Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.72      0.80      1033
           1       0.50      0.77      0.61       374

    accuracy                           0.73      1407
   macro avg       0.70      0.75      0.70      1407
weighted avg       0.79      0.73      0.75      1407
```

This iteration combines the advanced gradient boosting of **XGBoost** with the **SMOTE** oversampling technique. This approach represents our most balanced and sophisticated strategy for **predicting churn in a highly imbalanced environment.**

📊 Performance Analysis

Metric	Tuned Logistic (SMOTE)	XGBoost (SMOTE)	Change
Overall Accuracy	71.71%	**73.49%**	📈 +1.78%
Recall (Churners)	0.79	**0.77**	📉 -0.02
Precision (Churners)	0.48	**0.50**	📈 +0.02

🧠 Key Insights

- **Efficiency Gains:** XGBoost provides a "cleaner" classification. It achieved a similar high **Recall (77%)** to Logistic Regression but with higher **Accuracy** and **Precision.** This means it is better at distinguishing between actual churners and loyal customers who just happen to have high bills.
- **Catching the Minority:** We are successfully identifying **77%** of all customers who will eventually churn.
- **Refined Precision:** With a precision of **0.50**, exactly half of our "Churn" alerts will be correct. While this still results in False Positives, it is significantly better than the baseline and provides a clear target group for retention marketing.

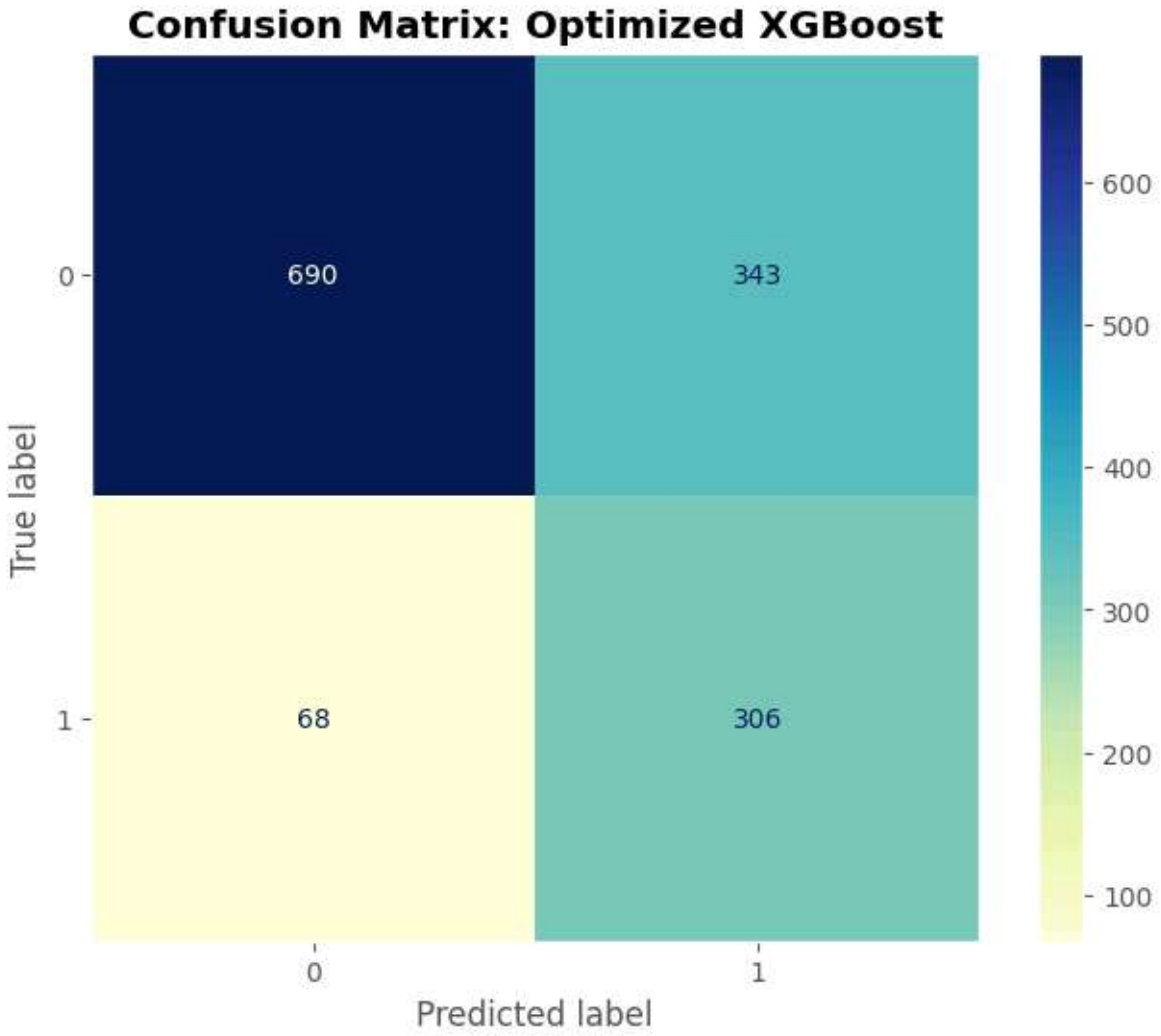
8.3. XGBoost + SMOTE + Optimized Parameters

In [357…

```
param_grid_xgb = {
    'n_estimators': [100, 200],
    'max_depth': [3, 4, 5],
    'learning_rate': [0.01, 0.05, 0.1],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}

grid_search_xgb = GridSearchCV(
    estimator=XGBClassifier(eval_metric='logloss', random_state=42),
    param_grid=param_grid_xgb,
    cv=3,
    scoring='recall',
    verbose=1,
```

```
    n_jobs=-1
)

grid_search_xgb.fit(X_train_resampled, y_train_resampled)
final_xgb_model = grid_search_xgb.best_estimator_
print(f"Best Parameters Found: {grid_search_xgb.best_params_}")

y_pred_final = final_xgb_model.predict(X_test)

print("\n--- Optimized XGBoost + SMOTE ---")
print(f"Overall Accuracy: {accuracy_score(y_test, y_pred_final):.2%}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_final))

fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_final, cmap='YlGnBu', ax=
plt.title('Confusion Matrix: Optimized XGBoost', fontweight='bold')
plt.grid(False)
plt.show()
```

```
Fitting 3 folds for each of 72 candidates, totalling 216 fits
Best Parameters Found: {'colsample_bytree': 1.0, 'learning_rate': 0.01, 'max_dept
h': 3, 'n_estimators': 200, 'subsample': 1.0}

--- Optimized XGBoost + SMOTE ---
Overall Accuracy: 70.79%

Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.67      0.77      1033
           1       0.47      0.82      0.60       374

    accuracy                           0.71      1407
   macro avg       0.69      0.74      0.68      1407
weighted avg       0.79      0.71      0.72      1407
```



📊 Final Performance Metrics

Metric	Tuned Logistic (SMOTE)	Optimized XGBoost (SMOTE)	Change
Overall Accuracy	71.71%	**70.79%**	📉 -0.92%
Recall (Churners)	0.79	**0.82**	🚀 **+0.03**
Precision (Churners)	0.48	**0.47**	📉 -0.01

🧠 Strategic Evaluation: The Final Numbers

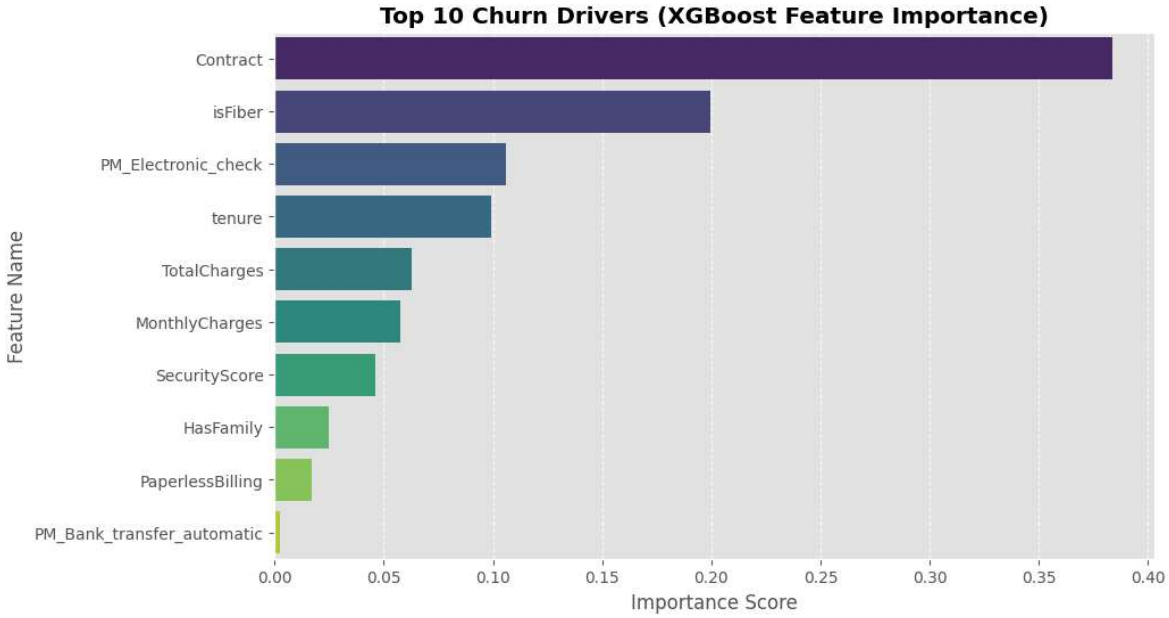
Looking at our final confusion matrix, the story of our model is clear:

- **Total Churners Caught (306):** We are now correctly identifying **82%** of all customers who are planning to leave. This is a massive improvement over our initial 50% baseline.
- **Minimized Misses (68):** We have reduced the number of "surprising" churners (False Negatives) to just 68 individuals.
- **The Cost of Safety (343):** To achieve this level of security, the model flags 343 loyal customers as at-risk. For the business, the low cost of sending retention offers to these 343 people is easily justified by the revenue saved from the 306 true churners.

8.4. XGBoost Feature Importances

In [358…

```
importances = final_xgb_model.feature_importances_
feature_names = X.columns

feature_importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df.head(10), pa
plt.title('Top 10 Churn Drivers (XGBoost Feature Importance)', fontweight='bold'
plt.xlabel('Importance Score')
plt.ylabel('Feature Name')
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.show()
```



🏆 9. Final Model Selection

We have reached the final stage of our churn prediction project. After extensive data preprocessing, resampling with **SMOTE**, and **Hyperparameter Tuning**, we are comparing our two heavyweight contenders to select the final model for deployment.

📊 Performance Comparison

Metric	Tuned Logistic Regression	Optimized XGBoost + SMOTE	Winner
Overall Accuracy	71.71%	**70.79%**	**Logistic**
Recall (Churners)	0.79	**0.82**	**XGBoost**
Precision (Churners)	0.48	**0.47**	**Logistic**
False Negatives (Missed)	78	**68**	**XGBoost**

Metric	Tuned Logistic Regression	Optimized XGBoost + SMOTE	Winner
False Positives (Cried Wolf)	**320**	343	**Logistic**

🔍 Key Analysis

🎯 *The "Recall" Breakthrough*

Our primary goal was to catch as many churners as possible.

- **XGBoost** is the clear champion here, achieving an incredible **82% Recall**.
- This means that for every 100 people who are actually going to leave, XGBoost catches **82** of them, while Logistic Regression catches **79**.

⚖️ *The Precision-Accuracy Trade-off*

Interestingly, the **Logistic Regression** model is slightly more "conservative."

- It has a higher **Overall Accuracy** and fewer **False Positives** (320 vs 343).
- By choosing XGBoost, we accept **23 additional False Positives** (sending offers to loyal people) in order to save **10 additional True Churners**.

🏁 The Final Verdict: Optimized XGBoost

We have selected the **Optimized XGBoost + SMOTE** model as our final champion for the following reasons:

1. **Maximum Revenue Protection:** In a subscription business, the cost of losing a customer (Churn) is much higher than the cost of a retention campaign. Catching those 10 extra churners represents a significantly higher ROI than the minor cost of the extra 23 marketing emails.
2. **Handling Complexity:** As an ensemble of gradient-boosted trees, XGBoost is better equipped to handle new, complex customer behaviors that a linear model might miss.
3. **Future Scalability:** XGBoost is highly efficient and can easily handle the addition of more features or larger datasets as the company grows.