SyriaTel Customer Churn Prediction

1. Introduction & Business Understanding

Problem Statement:

SyriaTel wants to proactively identify customers who are likely to churn (cancel service) so they can offer retention incentives and reduce revenue loss.

Stakeholder:
SyriaTel Retention Team / Marketing Department

Objective:
Build a binary classification model that predicts whether a customer will churn in the next billing cycle.

Success Criteria:

- High recall on churners (catch at least 80% of true churners).
- Precision ≥ 60% to minimize wasted retention offers.

2. Data Loading & Understanding

```
# Importing standard libraries
import pandas as pd
import numpy as np

# Importing visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Importing modeling libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score, accuracy_score

from sklearn.model_selection import GridSearchCV

from statsmodels.stats.outliers_influence import
```

```
variance_inflation_factor
from statsmodels.tools.tools import add_constant

# Loading the data
df = pd.read_csv('data/syriatel_churn.csv', sep=',', header=0)

# previewing the first 5 rows
df.head()
```

```
  state  account length  area code phone number international plan  \
0    KS             128        415     382-4657                 no
1    OH             107        415     371-7191                 no
2    NJ             137        415     358-1921                 no
3    OH              84        408     375-9999                yes
4    OK              75        415     330-6626                yes

  voice mail plan  number vmail messages  total day minutes  total day
calls  \
0             yes                     25              265.1
110
1             yes                     26              161.6
123
2              no                      0              243.4
114
3              no                      0              299.4
71
4              no                      0              166.7
113

   total day charge  ...  total eve calls  total eve charge  \
0             45.07  ...               99             16.78
1             27.47  ...              103             16.62
2             41.38  ...              110             10.30
3             50.90  ...               88              5.26
4             28.34  ...              122             12.61

   total night minutes  total night calls  total night charge  \
0                244.7                 91               11.01
1                254.4                103               11.45
2                162.6                104                7.32
3                196.9                 89                8.86
4                186.9                121                8.41

   total intl minutes  total intl calls  total intl charge  \
0                10.0                 3               2.70
1                13.7                 3               3.70
2                12.2                 5               3.29
3                 6.6                 7               1.78
4                10.1                 3               2.73
```

```
   customer service calls  churn
0                       1  False
1                       1  False
2                       0  False
3                       2  False
4                       3  False

[5 rows x 21 columns]
```

```
# # previewing the last 5 rows
df.tail()
```

```
     state  account length  area code phone number international plan
\
3328    AZ             192        415     414-4276                 no

3329    WV              68        415     370-3271                 no

3330    RI              28        510     328-8230                 no

3331    CT             184        510     364-6381                yes

3332    TN              74        415     400-4344                 no


     voice mail plan  number vmail messages  total day minutes  \
3328             yes                     36              156.2
3329              no                      0              231.1
3330              no                      0              180.8
3331              no                      0              213.8
3332             yes                     25              234.4

      total day calls  total day charge  ...  total eve calls  \
3328               77             26.55  ...              126
3329               57             39.29  ...               55
3330              109             30.74  ...               58
3331              105             36.35  ...               84
3332              113             39.85  ...               82

      total eve charge  total night minutes  total night calls  \
3328             18.32                279.1                 83
3329             13.04                191.3                123
3330             24.55                191.9                 91
3331             13.57                139.2                137
3332             22.60                241.4                 77

      total night charge  total intl minutes  total intl calls  \
3328               12.56                 9.9                 6
3329                8.61                 9.6                 4
3330                8.64                14.1                 6
3331                6.26                 5.0                10
```

```
3332                 10.86                  13.7                 4

      total intl charge  customer service calls  churn
3328               2.67                       2  False
3329               2.59                       3  False
3330               3.81                       2  False
3331               1.35                       2  False
3332               3.70                       0  False

[5 rows x 21 columns]
```

Initial inspection

```
# getting the shape
df.shape
print(f"Rows:{df.shape[0]}, Columns: {df.shape[1]}")

Rows:3333, Columns: 21

# Breif info on the data
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 21 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   state                   3333 non-null   object 
 1   account length          3333 non-null   int64  
 2   area code               3333 non-null   int64  
 3   phone number            3333 non-null   object 
 4   international plan      3333 non-null   object 
 5   voice mail plan         3333 non-null   object 
 6   number vmail messages   3333 non-null   int64  
 7   total day minutes       3333 non-null   float64
 8   total day calls         3333 non-null   int64  
 9   total day charge        3333 non-null   float64
 10  total eve minutes       3333 non-null   float64
 11  total eve calls         3333 non-null   int64  
 12  total eve charge        3333 non-null   float64
 13  total night minutes     3333 non-null   float64
 14  total night calls       3333 non-null   int64  
 15  total night charge      3333 non-null   float64
 16  total intl minutes      3333 non-null   float64
 17  total intl calls        3333 non-null   int64  
 18  total intl charge       3333 non-null   float64
 19  customer service calls  3333 non-null   int64  
 20  churn                   3333 non-null   bool   
dtypes: bool(1), float64(8), int64(8), object(4)
memory usage: 524.2+ KB
```

```
# columns
df.columns
```

```
Index(['state', 'account length', 'area code', 'phone number',
       'international plan', 'voice mail plan', 'number vmail
messages',
       'total day minutes', 'total day calls', 'total day charge',
       'total eve minutes', 'total eve calls', 'total eve charge',
       'total night minutes', 'total night calls', 'total night
charge',
       'total intl minutes', 'total intl calls', 'total intl charge',
       'customer service calls', 'churn'],
      dtype='object')
```

Initial Data Inspection Summary

- **Dataset Overview**
  - 3,333 customer records, 21 attributes
  - No missing values; memory footprint ~525 KB
- **Data Types**
  - 8 integer features (e.g., `account length`, `total day calls`)
  - 8 float features (e.g., `total day minutes`, `total night charge`)
  - 4 categorical features (`state`, `international plan`, etc.)
  - 1 boolean target (`churn`)
- **Key Attributes**
  - **Usage Metrics:** Day, evening, night, and international minutes & charges
  - **Subscription Details:** Account length, contract type, plan indicators
  - **Customer Interaction:** Customer service call counts
- **Target Variable**
  - `churn` (True/False): Indicates service discontinuation

Executive Insight:
The dataset is complete and well-structured, with balanced representation across usage, subscription, and interaction metrics. No immediate data quality issues detected—ready for in-depth exploratory analysis.

Data Dictionary

Below is a list of each feature in the SyriaTel churn dataset and its definition:

- **state**
  Two-letter U.S. state code where the customer resides.

- **account length**
  Number of months the customer has been active on the network.
- **area code**
  Three-digit telephone area code.
- **phone number**
  Customer’s unique phone identifier (de-identified string).
- **international plan**
  “yes” / “no” – whether the customer subscribes to an international calling plan.
- **voice mail plan**
  “yes” / “no” – whether the customer has an active voicemail service.
- **number vmail messages**
  Count of voicemail messages sent or received by the customer.
- **total day minutes**
  Total minutes of calls made during daytime hours.
- **total day calls**
  Total number of calls placed during daytime hours.
- **total day charge**
  Dollar amount billed for daytime calls.
- **total eve minutes**
  Total minutes of calls made during evening hours.
- **total eve calls**
  Total number of calls placed during evening hours.
- **total eve charge**
  Dollar amount billed for evening calls.
- **total night minutes**
  Total minutes of calls made during nighttime hours.
- **total night calls**
  Total number of calls placed during nighttime hours.
- **total night charge**
  Dollar amount billed for nighttime calls.
- **total intl minutes**
  Total minutes of international calls.
- **total intl calls**
  Total number of international calls placed.

- **total intl charge**
  Dollar amount billed for international calls.

- **customer service calls**
  Number of calls the customer made to customer service.

- **churn**
  Boolean indicator (True/False) of whether the customer discontinued service ("churned").

3. Exploratory Data Analysis (EDA)

In this section, we'll explore data quality, distributions, and relationships to guide our feature engineering and modeling.

3.1 Missing values & Duplicates

```
df.isna().sum()
```

```
state                     0
account length            0
area code                 0
phone number              0
international plan        0
voice mail plan           0
number vmail messages     0
total day minutes         0
total day calls           0
total day charge          0
total eve minutes         0
total eve calls           0
total eve charge          0
total night minutes       0
total night calls         0
total night charge        0
total intl minutes        0
total intl calls          0
total intl charge         0
customer service calls    0
churn                     0
dtype: int64
```

Missing Values Check

- **Result:** All 21 features have 0 missing values.
- **Implication:** No imputation or row deletion required—dataset is complete and ready for further analysis.

Identifying duplicate customer IDs

```
duplicates = df.duplicated(subset='phone number').sum()
print(f"Duplicate customer records: {duplicates}")
```

```
Duplicate customer records: 0
```

Duplicate Records Check

- **Method:** Checked for duplicate entries using the `phone number` as the unique customer identifier.
- **Result:** 0 duplicate records found.
- **Implication:** No duplicates to remove—each row represents a unique customer.

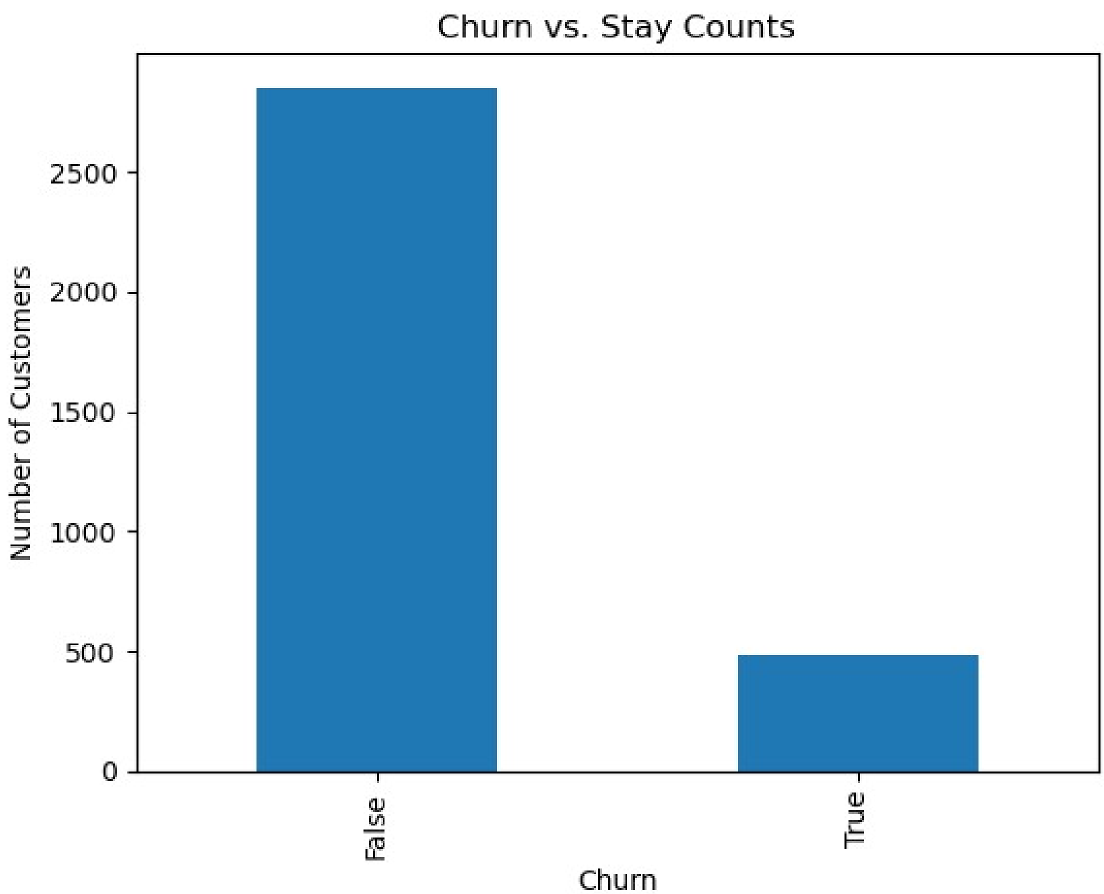
3.2 Target Distribution

```
# computing churn counts
churn_counts = df['churn'].value_counts()
print(churn_counts)
```

```
churn
False    2850
True      483
Name: count, dtype: int64
```

- **Churn Value Counts:**
  - `False` (Stayed): 2,850 customers
  - `True` (Churned): 483 customers
- **Interpretation:**
  - Approximately 14.5% of customers have churned (`483 / 3333`), indicating a moderate class imbalance that may need to be addressed in modeling.

```
# plotting churn rate
churn_counts.plot(kind='bar', title='Churn vs. Stay Counts')
plt.xlabel('Churn')
plt.ylabel('Number of Customers');
```




```
# calculating % churned
(churn_counts / df.shape[0]) * 100

churn
False    85.508551
True     14.491449
Name: count, dtype: float64
```

- **Churn Proportions:**
  - Stayed (`False`): ~85.5%

  - Churned (`True`): ~14.5%
- **Business Implication:**
  With only about 15% of customers churning, the dataset exhibits moderate imbalance. When training models, we should ensure this imbalance doesn't bias results—options include stratified train/test splits, using specialized metrics (e.g., recall, F1-score), or applying resampling techniques.

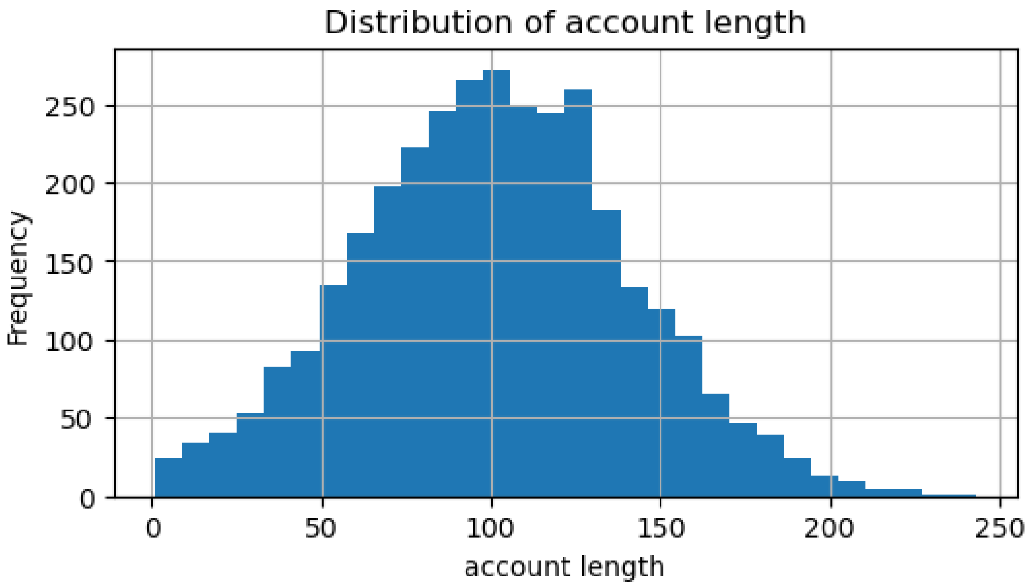
3.3 Univariate Analysis

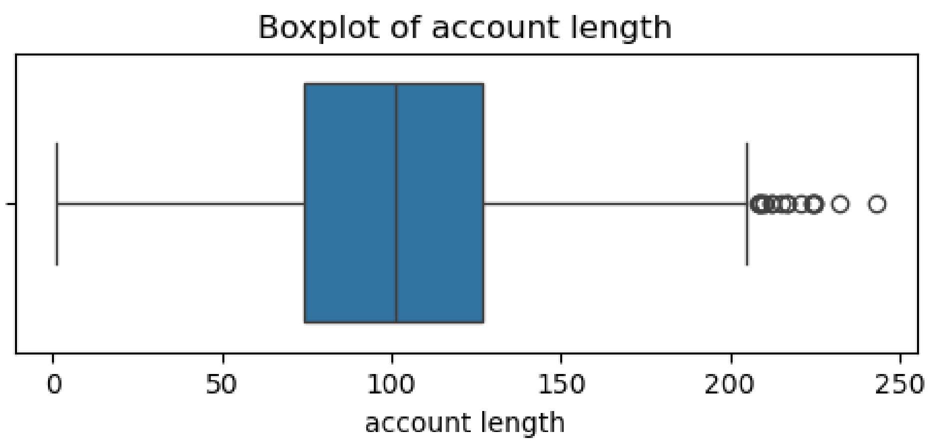
3.3.1 Numerical Features

For each numeric column, we will:

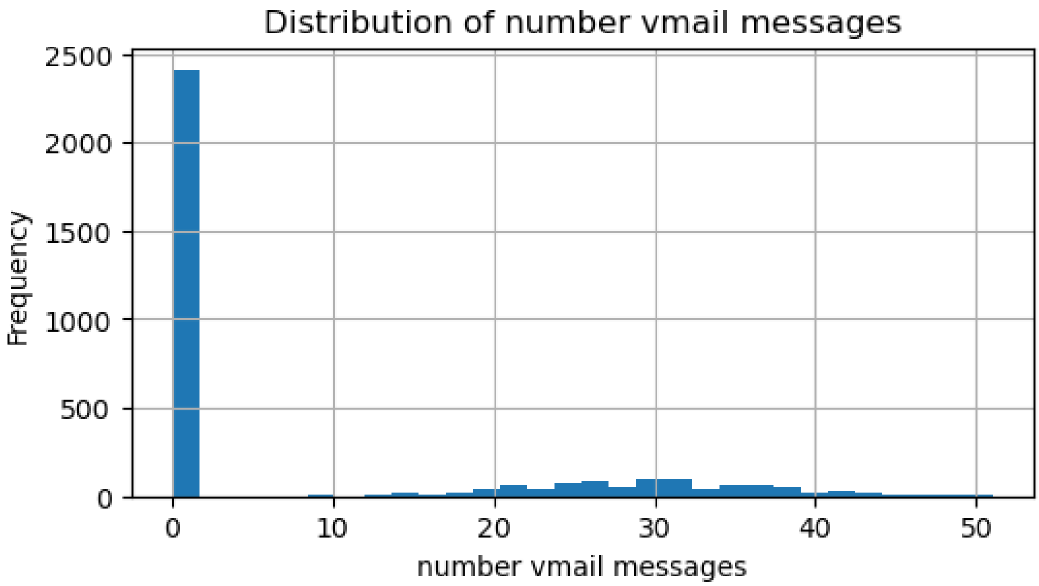
1. Plot a histogram to see the distribution.
2. Plot a boxplot to identify outliers.
3. Display basic descriptive statistics.

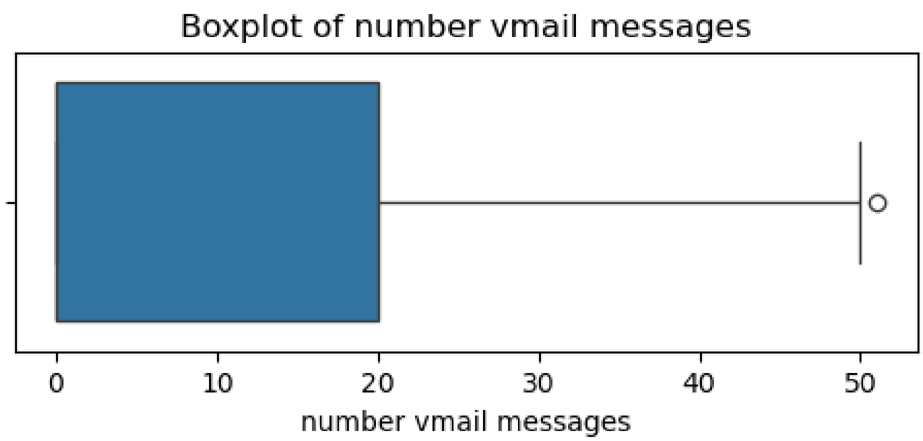
```
# List of numeric columns to analyze
numeric_cols = [
    'account length', 'number vmail messages',
    'total day minutes', 'total day calls', 'total day charge',
    'total eve minutes', 'total eve calls', 'total eve charge',
    'total night minutes', 'total night calls', 'total night charge',
    'total intl minutes', 'total intl calls', 'total intl charge',
    'customer service calls'
]

for col in numeric_cols:
    # Histogram
    plt.figure(figsize=(6, 3))
    df[col].hist(bins=30)
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.show()

    # Boxplot
    plt.figure(figsize=(6, 2))
    sns.boxplot(x=df[col])
    plt.title(f'Boxplot of {col}')
    plt.show()

    # Descriptive statistics
    print(f"--- {col} ---")
    display(df[col].describe())
    print("\n")
```



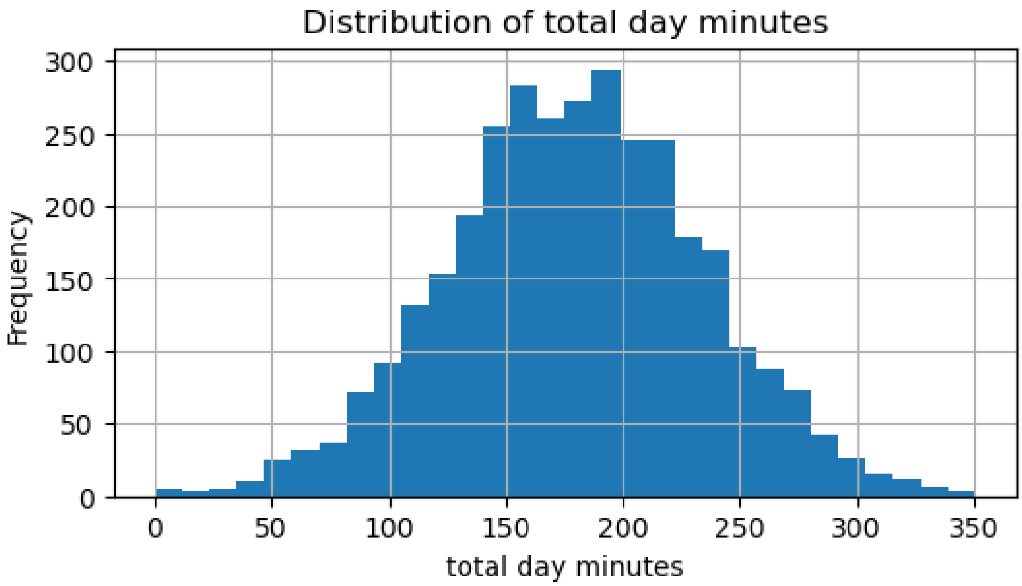


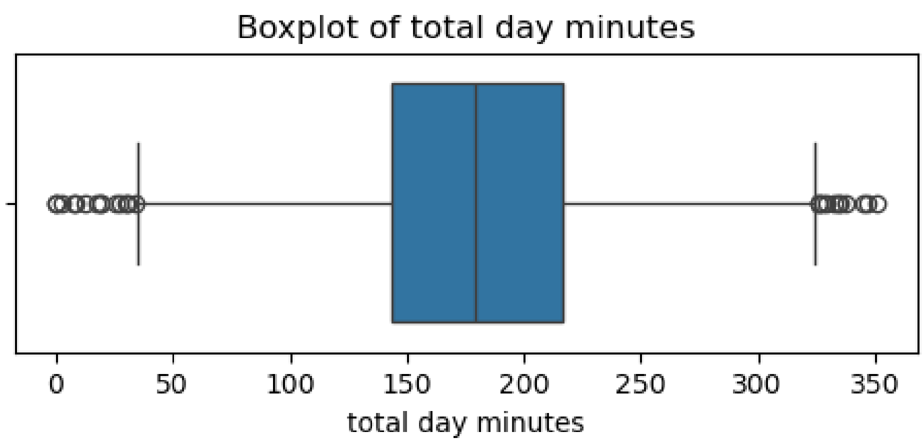

```
--- account length ---

count    3333.000000
mean      101.064806
std        39.822106
min         1.000000
25%        74.000000
50%       101.000000
75%       127.000000
max       243.000000
Name: account length, dtype: float64
```



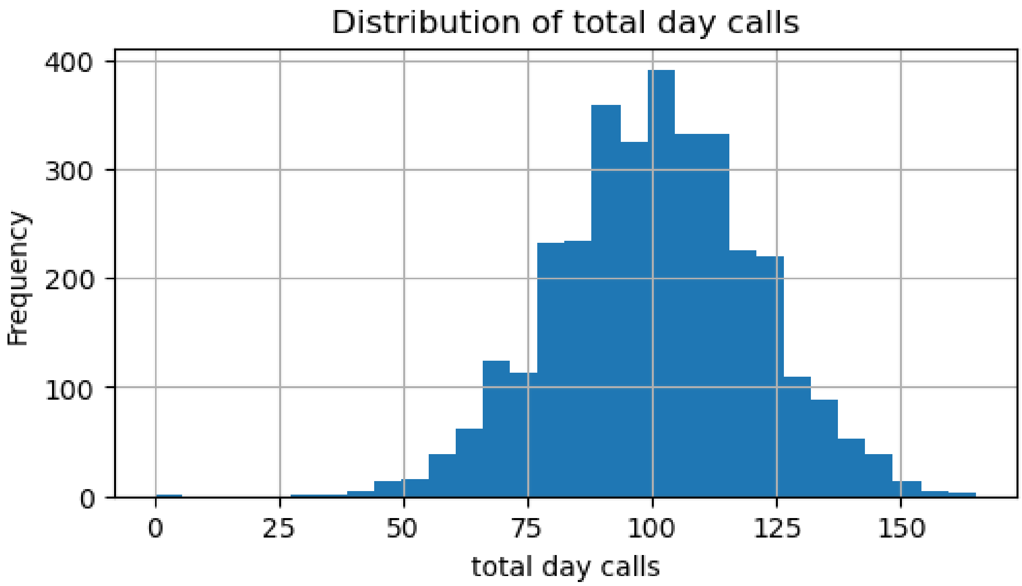


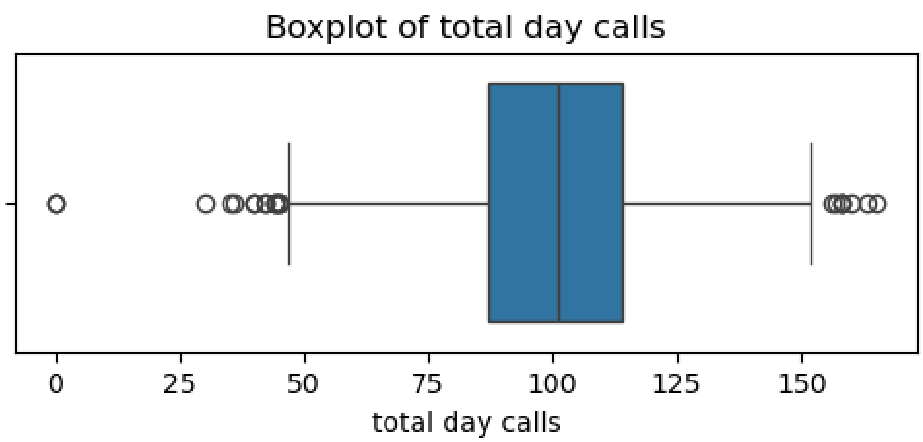

```
--- number vmail messages ---

count    3333.000000
mean        8.099010
std        13.688365
min         0.000000
25%         0.000000
50%         0.000000
75%        20.000000
max        51.000000
Name: number vmail messages, dtype: float64
```



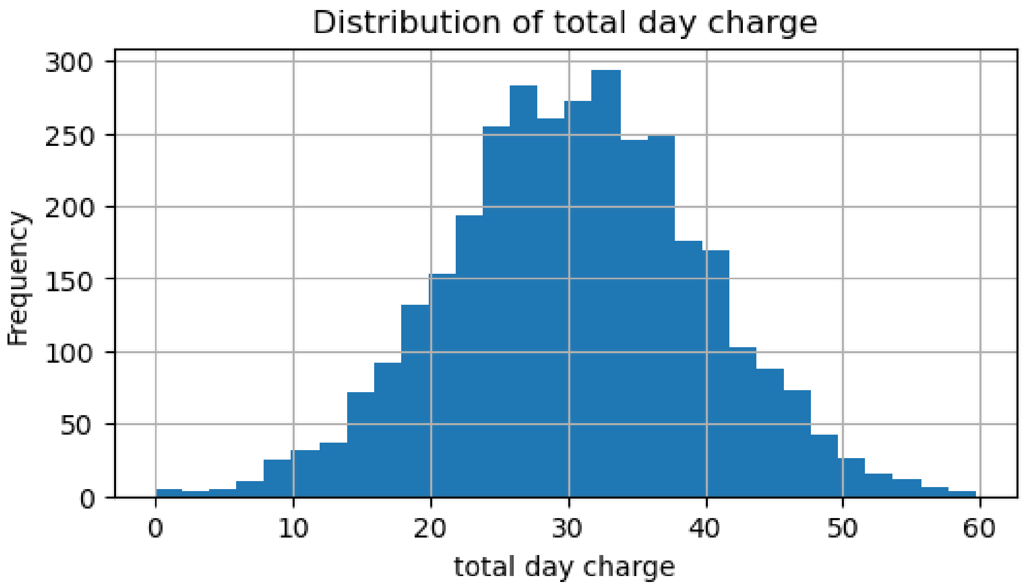


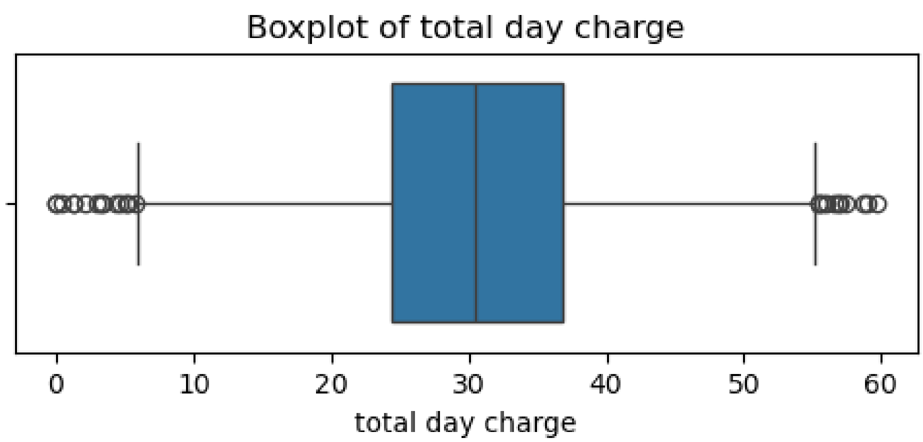

```
--- total day minutes ---

count    3333.000000
mean      179.775098
std        54.467389
min         0.000000
25%       143.700000
50%       179.400000
75%       216.400000
max       350.800000
Name: total day minutes, dtype: float64
```



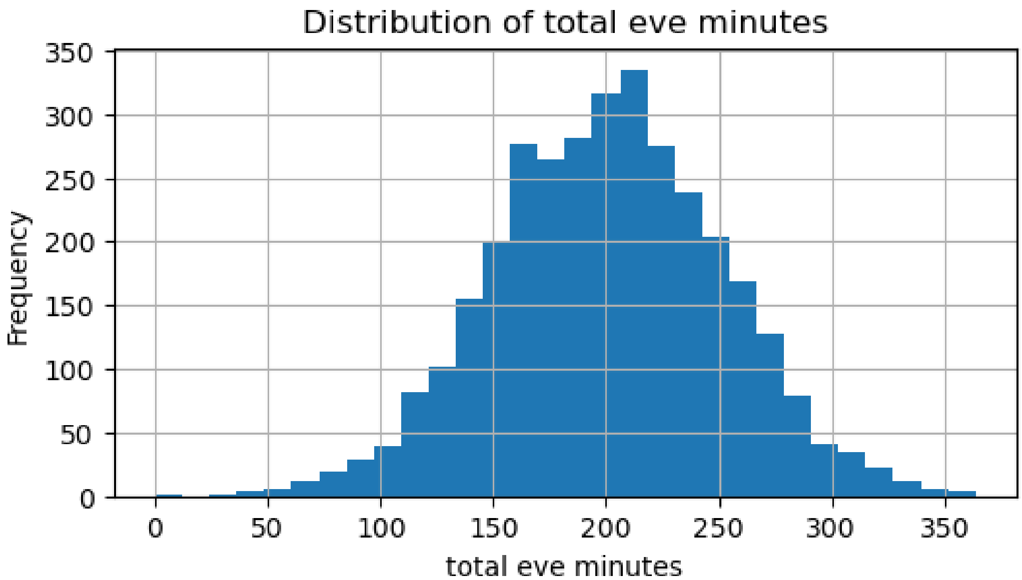


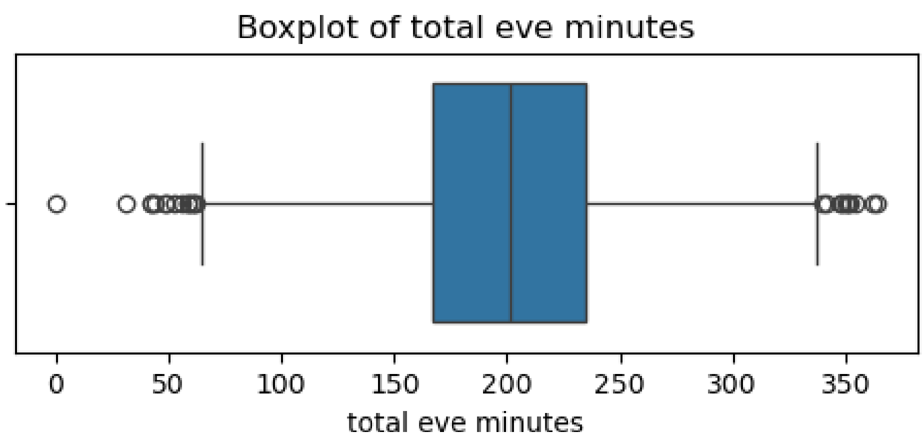

```
--- total day calls ---

count    3333.000000
mean      100.435644
std        20.069084
min         0.000000
25%        87.000000
50%       101.000000
75%       114.000000
max       165.000000
Name: total day calls, dtype: float64
```



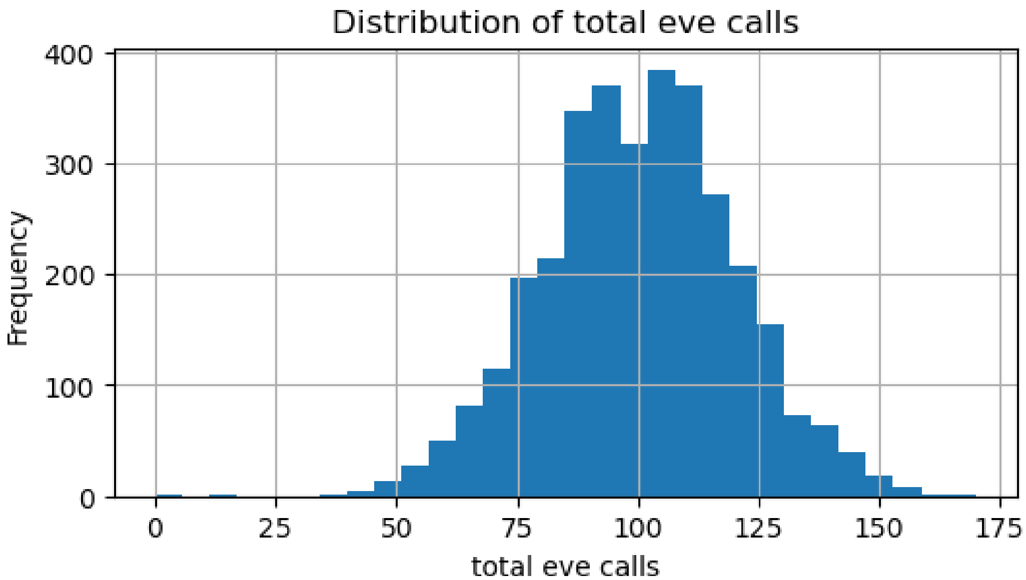


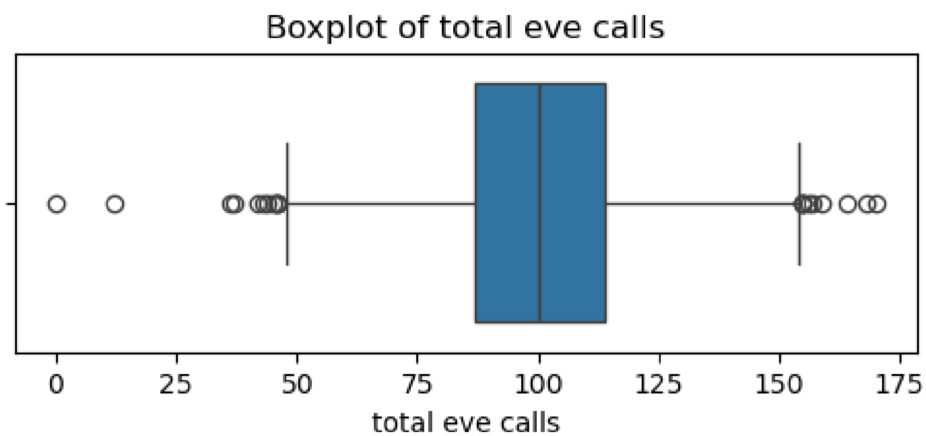

```
--- total day charge ---

count    3333.000000
mean       30.562307
std         9.259435
min         0.000000
25%        24.430000
50%        30.500000
75%        36.790000
max        59.640000
Name: total day charge, dtype: float64
```



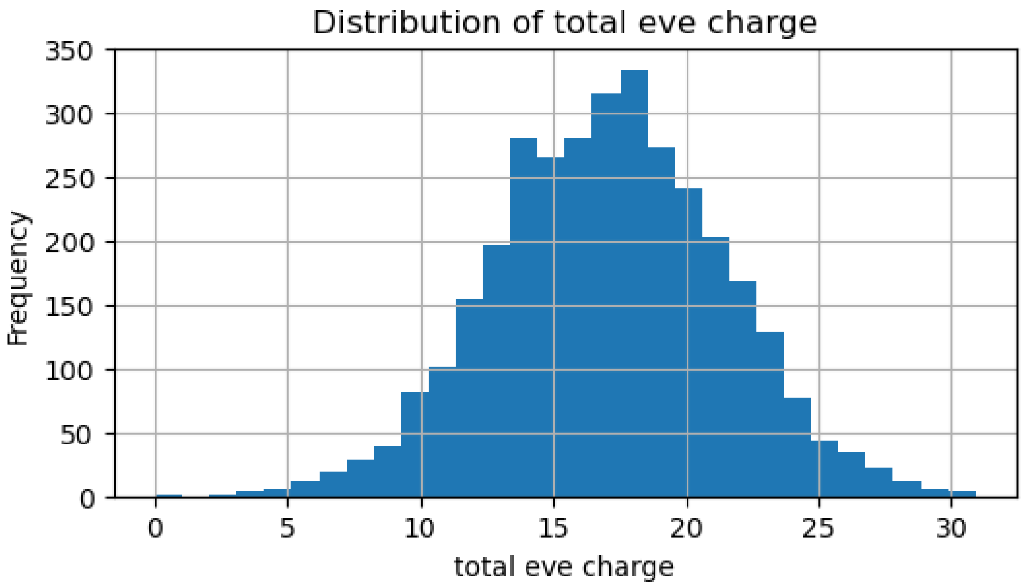


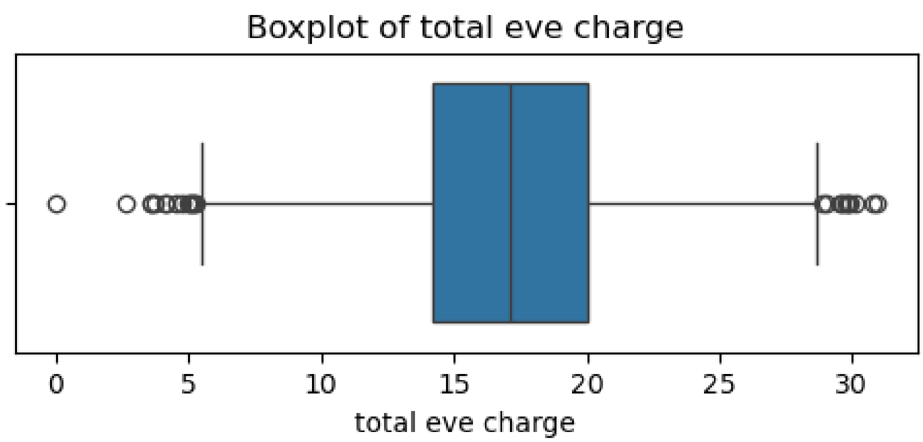

```
--- total eve minutes ---

count    3333.000000
mean      200.980348
std        50.713844
min         0.000000
25%       166.600000
50%       201.400000
75%       235.300000
max       363.700000
Name: total eve minutes, dtype: float64
```



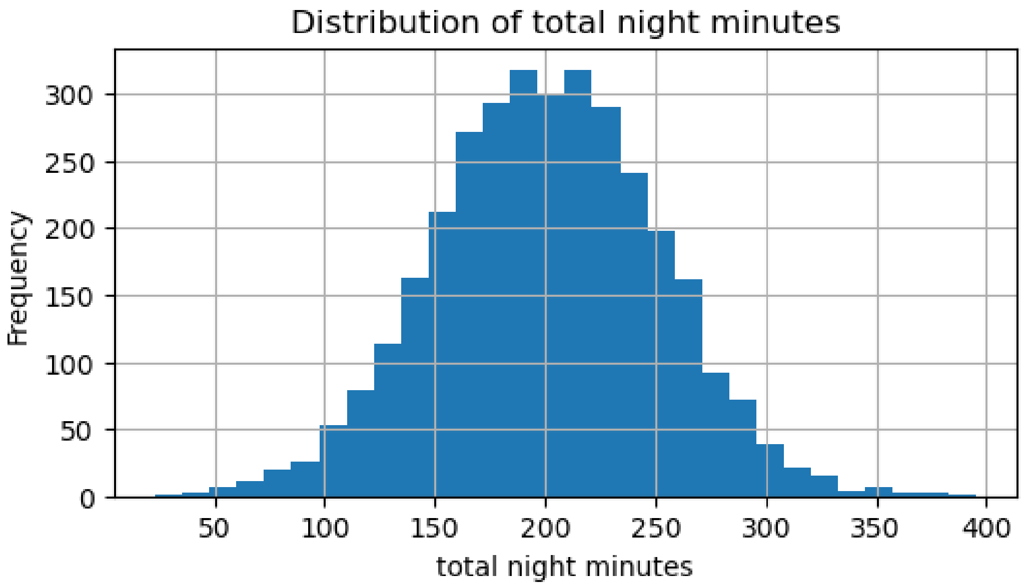


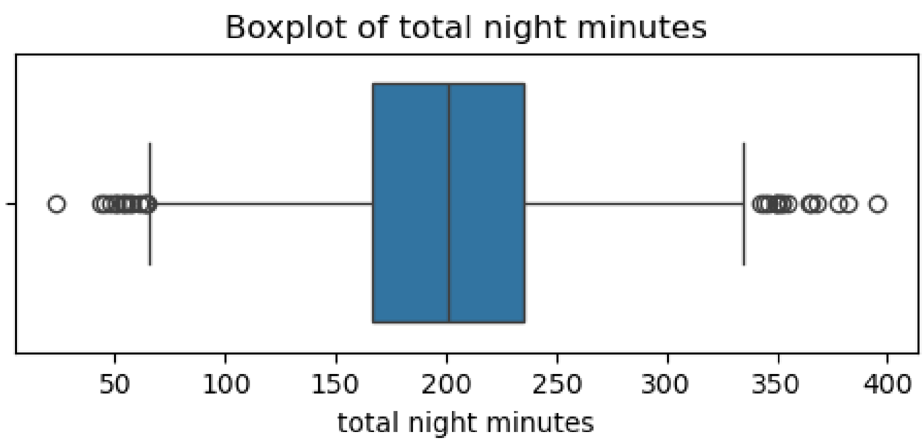

```
--- total eve calls ---

count    3333.000000
mean      100.114311
std        19.922625
min         0.000000
25%        87.000000
50%       100.000000
75%       114.000000
max       170.000000
Name: total eve calls, dtype: float64
```



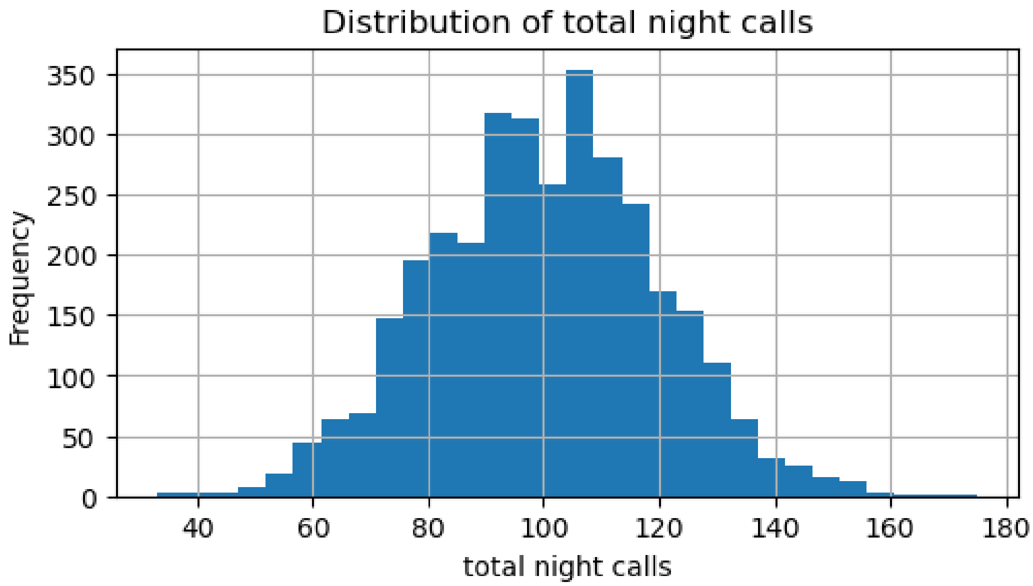


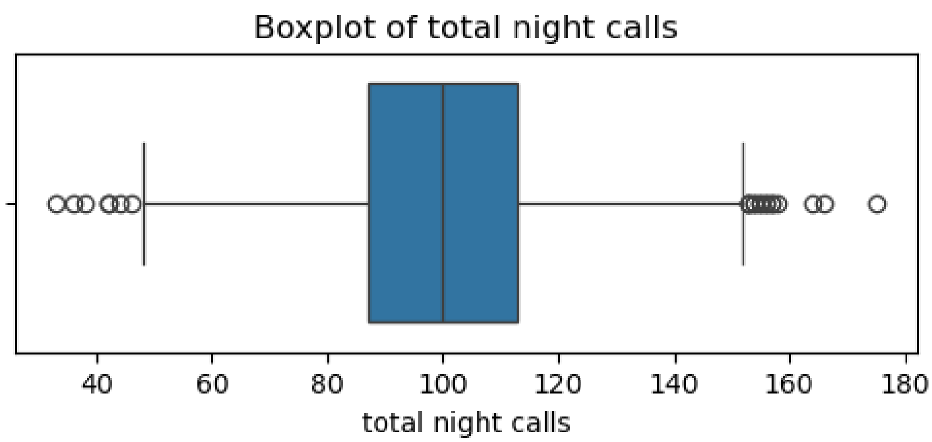

```
--- total eve charge ---

count    3333.000000
mean       17.083540
std         4.310668
min         0.000000
25%        14.160000
50%        17.120000
75%        20.000000
max        30.910000
Name: total eve charge, dtype: float64
```



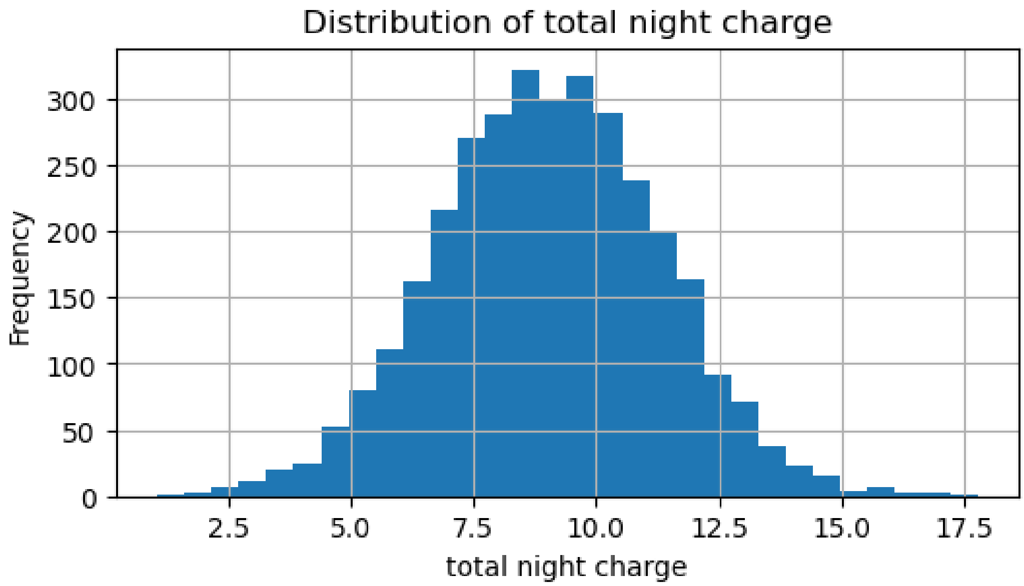


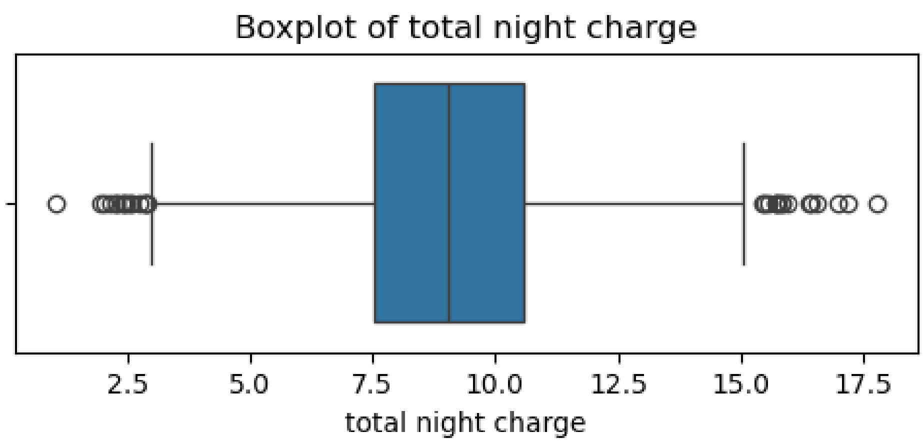

```
--- total night minutes ---

count    3333.000000
mean      200.872037
std        50.573847
min        23.200000
25%       167.000000
50%       201.200000
75%       235.300000
max       395.000000
Name: total night minutes, dtype: float64
```



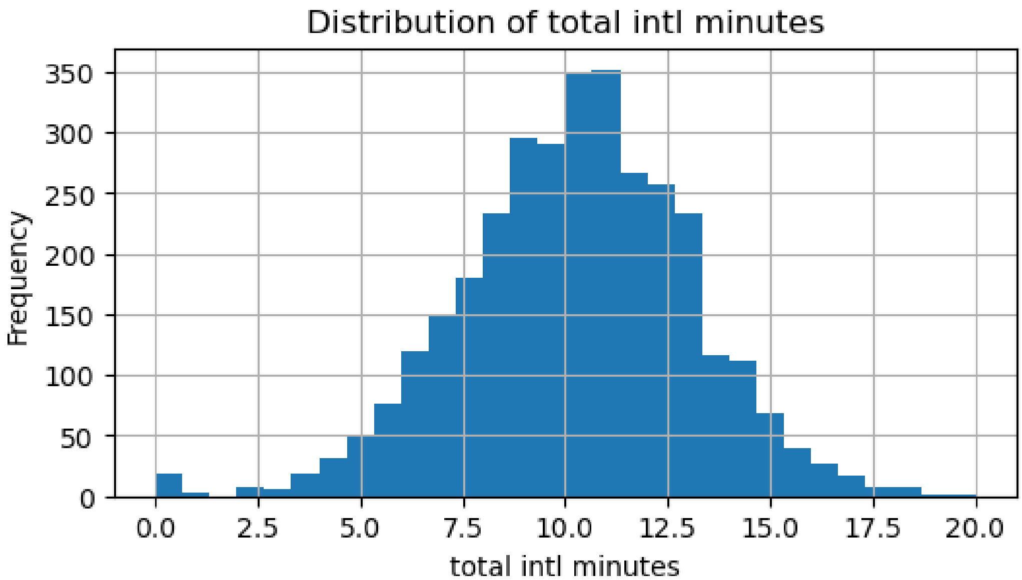


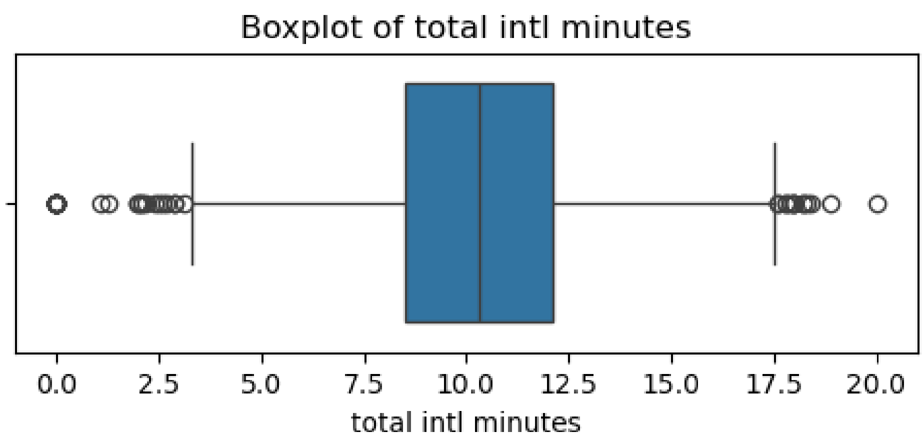

```
--- total night calls ---

count    3333.000000
mean      100.107711
std        19.568609
min        33.000000
25%        87.000000
50%       100.000000
75%       113.000000
max       175.000000
Name: total night calls, dtype: float64
```



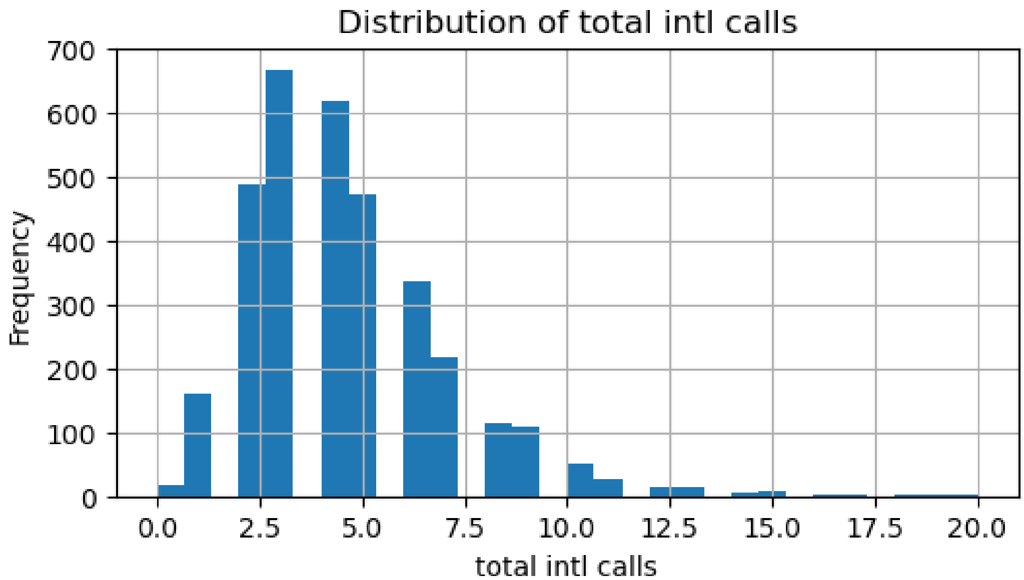


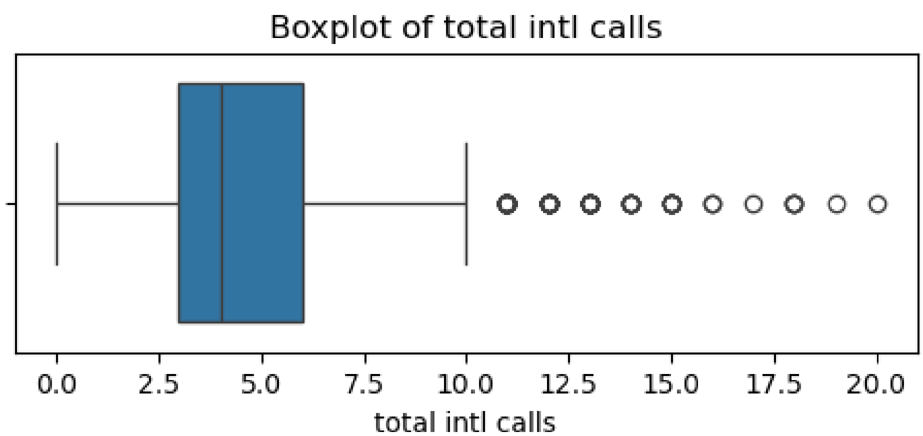

```
--- total night charge ---

count    3333.000000
mean        9.039325
std         2.275873
min         1.040000
25%         7.520000
50%         9.050000
75%        10.590000
max        17.770000
Name: total night charge, dtype: float64
```



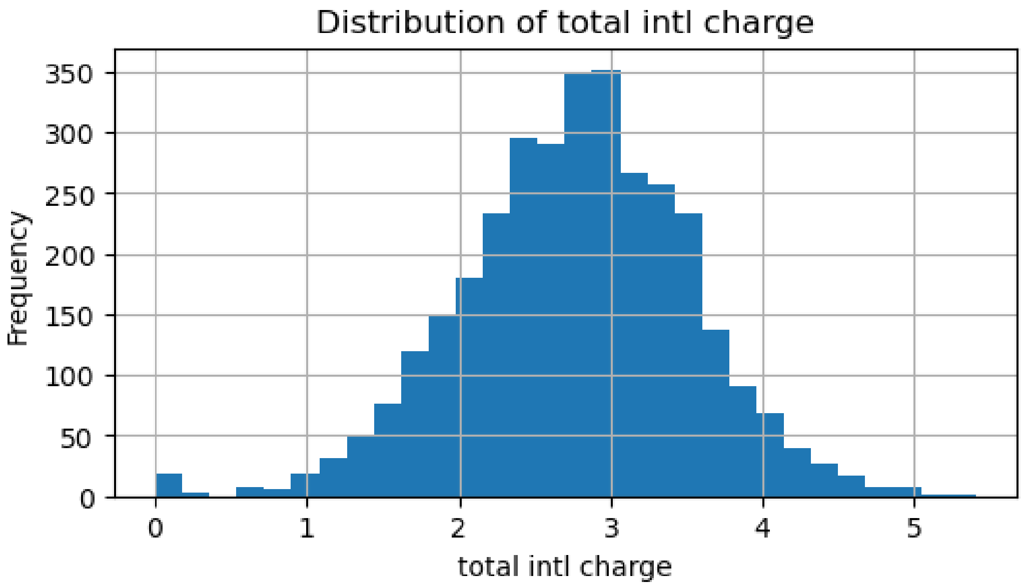


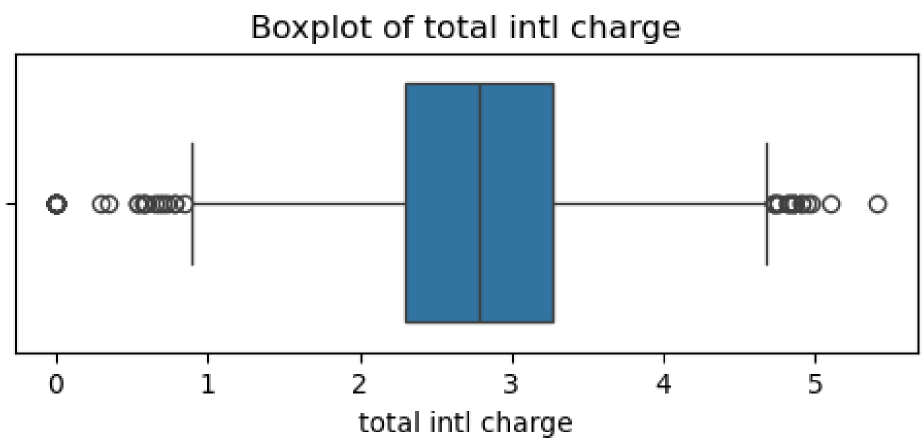

```
--- total intl minutes ---

count    3333.000000
mean       10.237294
std         2.791840
min         0.000000
25%         8.500000
50%        10.300000
75%        12.100000
max        20.000000
Name: total intl minutes, dtype: float64
```



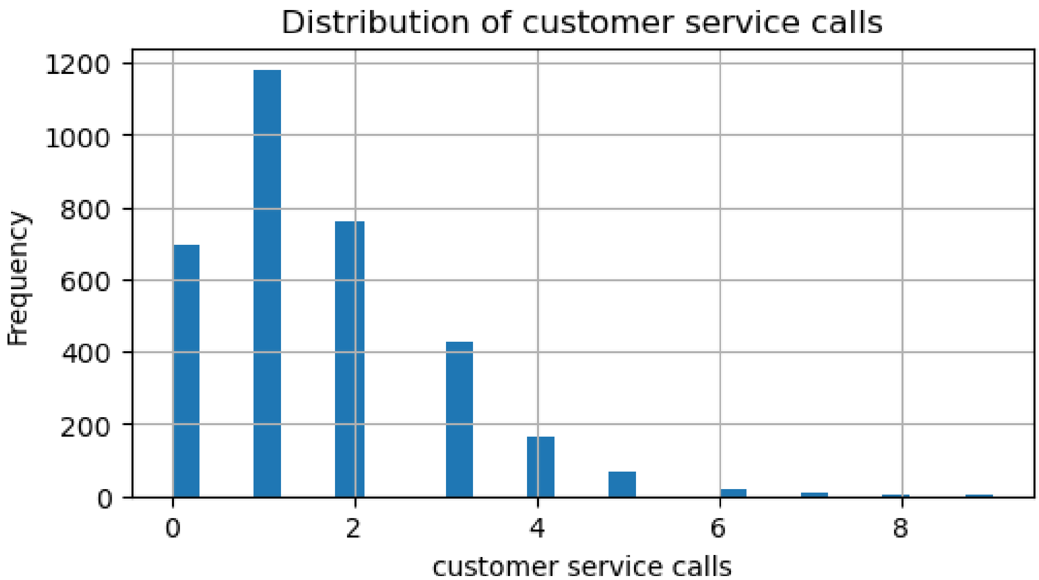


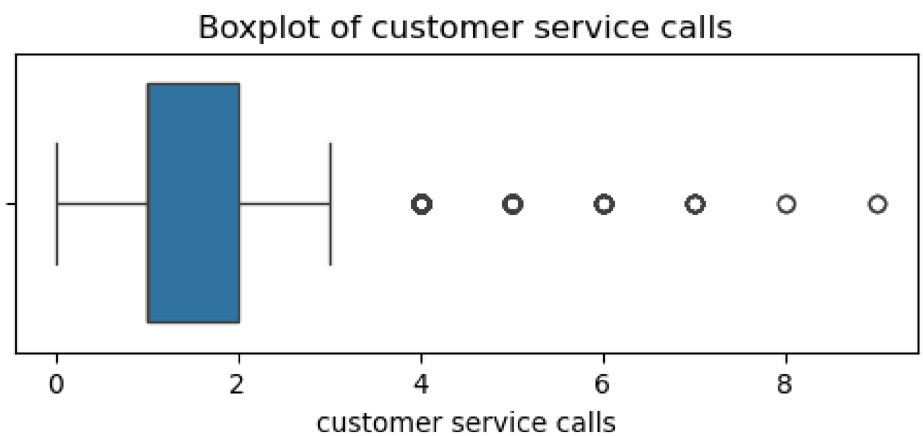

```
--- total intl calls ---

count    3333.000000
mean        4.479448
std         2.461214
min         0.000000
25%         3.000000
50%         4.000000
75%         6.000000
max        20.000000
Name: total intl calls, dtype: float64
```






```
--- total intl charge ---

count    3333.000000
mean        2.764581
std         0.753773
min         0.000000
25%         2.300000
50%         2.780000
75%         3.270000
max         5.400000
Name: total intl charge, dtype: float64
```






```
--- customer service calls ---

count    3333.000000
mean        1.562856
std         1.315491
min         0.000000
25%         1.000000
50%         1.000000
75%         2.000000
max         9.000000
Name: customer service calls, dtype: float64
```

3.3.1.1 Numerical Features – Distribution & Outlier Summary

- **Account Length**

- **Shape:** Approximately symmetric around 101 months (mean=101, median=101).
- **Outliers:** Values above ~180 months (mean ± 2×std = 101 ± 80) are rare.
- **Typical Range:** ~21 to 181 months.

- **Number of Voicemail Messages**
  - **Shape:** Highly right-skewed; 50% of customers have 0 messages (median=0).
  - **Outliers:** Customers with >35 messages (mean + 2×std ≈ 35) are uncommon.
  - **Typical Range:** 0 to 22 messages.
- **Total Day Minutes**
  - **Shape:** Moderately symmetric around 180 minutes (mean=180, median=179).
  - **Outliers:** Calls >288 minutes (mean + 2×std ≈ 288) are infrequent.
  - **Typical Range:** 71 to 288 minutes.
- **Total Day Calls**
  - **Shape:** Approximately normal around 100 calls (mean=100, median=101).
  - **Outliers:** >140 calls (mean + 2×std ≈ 140) are rare.
  - **Typical Range:** 60 to 140 calls.
- **Total Day Charge**
  - **Shape:** Mirrors day minutes, symmetric around $30.50.
  - **Outliers:** Charges >$49 (mean + 2×std ≈ $49) are uncommon.
  - **Typical Range:** $12 to $49.
- **Total Evening Minutes**
  - **Shape:** Centered around 201 minutes (mean=201, median=201).
  - **Outliers:** >302 minutes (mean + 2×std ≈ 302).
  - **Typical Range:** 100 to 302 minutes.
- **Total Evening Calls**
  - **Shape:** Near-normal around 100 calls.
  - **Outliers:** >140 calls.
  - **Typical Range:** 60 to 140 calls.
- **Total Evening Charge**
  - **Shape:** Symmetric around $17 (mean=17.08, median=17.12).

- **Outliers:** >\$26.7 (mean + 2×std).
- **Typical Range:** \$8.5 to \$26.7.

- **Total Night Minutes**
  - **Shape:** Centered around 201 minutes, slight right skew (min=23).
  - **Outliers:** >302 minutes.
  - **Typical Range:** 100 to 302 minutes.
- **Total Night Calls**
  - **Shape:** Normal around 100 calls, min=33 calls.
  - **Outliers:** >140 calls.
  - **Typical Range:** 60 to 140 calls.
- **Total Night Charge**
  - **Shape:** Symmetric around \$9 (mean=9.04, median=9.05).
  - **Outliers:** >\$13.6 (mean + 2×std).
  - **Typical Range:** \$4.5 to \$13.6.
- **Total International Minutes**
  - **Shape:** Mildly symmetric around 10.3 minutes.
  - **Outliers:** >15.8 minutes (mean + 2×std).
  - **Typical Range:** 4.7 to 15.8 minutes.
- **Total International Calls**
  - **Shape:** Right-skewed; median=4, mean=4.48.
  - **Outliers:** >9.4 calls.
  - **Typical Range:** 0 to 9.4 calls.
- **Total International Charge**
  - **Shape:** Symmetric around \$2.78.
  - **Outliers:** >\$4.27.
  - **Typical Range:** \$1.26 to \$4.27.
- **Customer Service Calls**
  - **Shape:** Right-skewed; median=1, mean=1.56.
  - **Outliers:** ≥4 calls (mean + 2×std ≈ 4.2).
  - **Typical Range:** 0 to 4 calls.

3.3.2 Categorical Features

For each categorical column, we will:

Show the value counts.

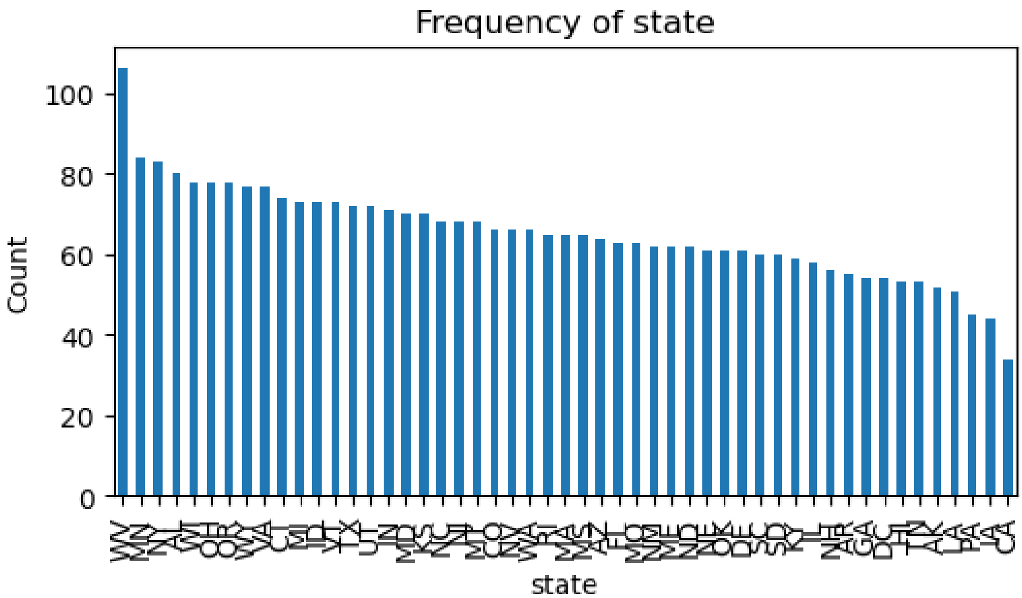
Plot a bar chart of frequencies.

```
# List of categorical columns to analyze
categorical_cols = ['state', 'area code', 'international plan', 'voice
mail plan']

for col in categorical_cols:
    # Value counts
    counts = df[col].value_counts()
    print(f"--- {col} ---")
    display(counts)

    # Bar plot
    plt.figure(figsize=(6, 3))
    counts.plot(kind='bar')
    plt.title(f'Frequency of {col}')
    plt.xlabel(col)
    plt.ylabel('Count')
    plt.show()

    print("\n")
```

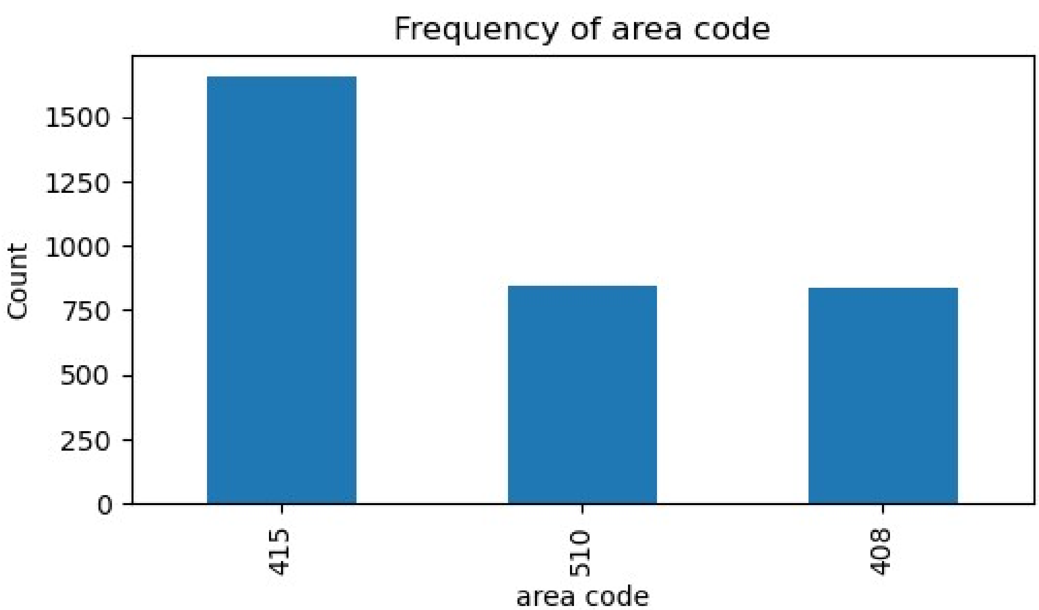
```
--- state ---
```

```
state
WV    106
MN     84
NY     83
AL     80
WI     78
OH     78
OR     78
WY     77
VA     77
CT     74
MI     73
ID     73
VT     73
TX     72
UT     72
IN     71
MD     70
KS     70
NC     68
NJ     68
```

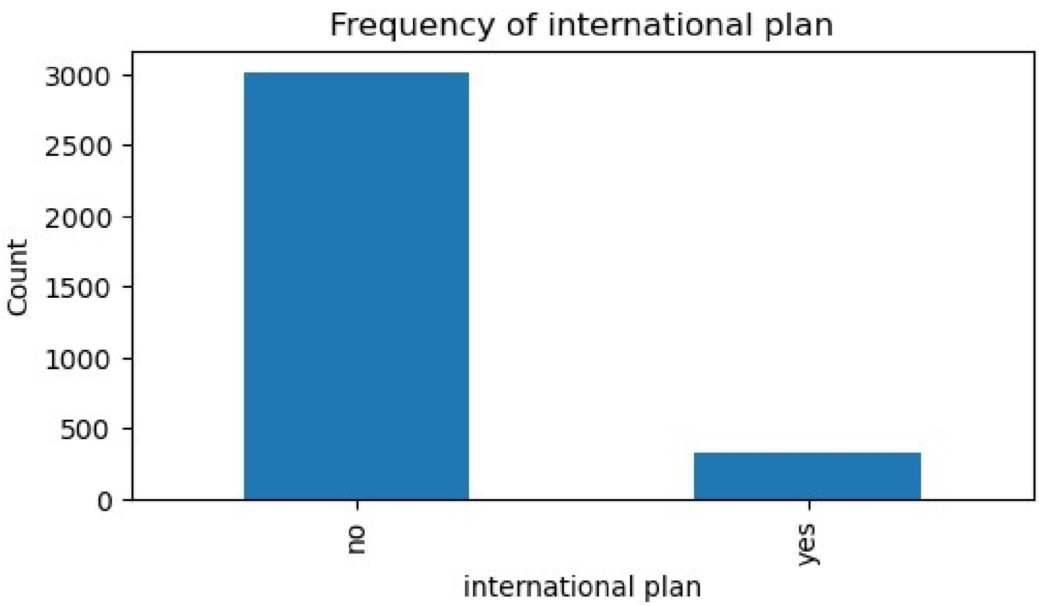
```
MT      68
CO      66
NV      66
WA      66
RI      65
MA      65
MS      65
AZ      64
FL      63
MO      63
NM      62
ME      62
ND      62
NE      61
OK      61
DE      61
SC      60
SD      60
KY      59
IL      58
NH      56
AR      55
GA      54
DC      54
HI      53
TN      53
AK      52
LA      51
PA      45
IA      44
CA      34
Name: count, dtype: int64
```



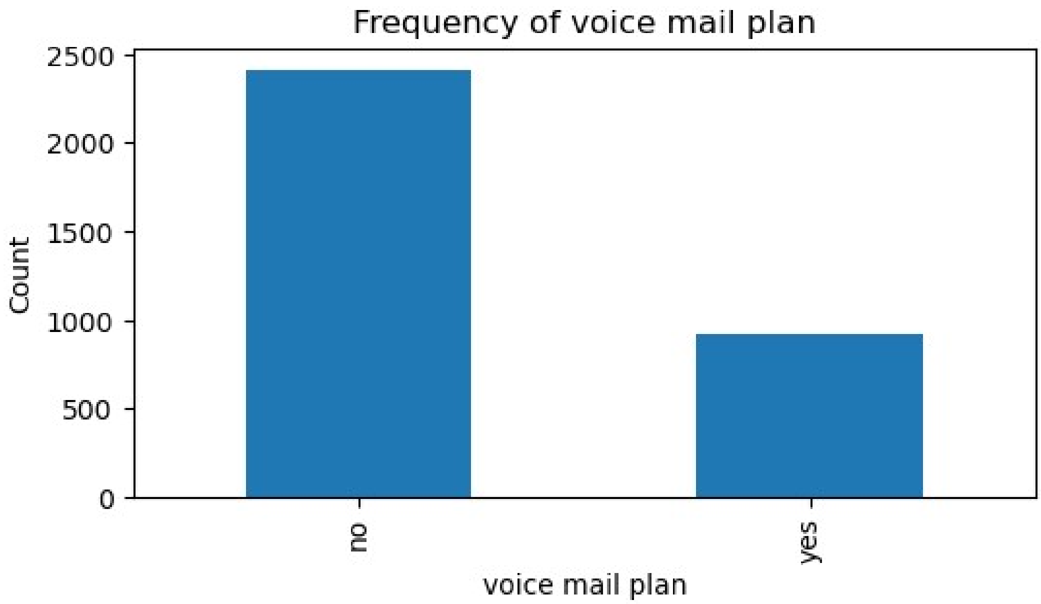

```
--- area code ---

area code
415    1655
510     840
408     838
Name: count, dtype: int64
```



```
--- international plan ---

international plan
no     3010
yes     323
Name: count, dtype: int64
```




```
--- voice mail plan ---

voice mail plan
no     2411
yes     922
Name: count, dtype: int64
```



3.3.2.2 Categorical Features – Frequency & Data Quality Summary

- **state**
  - **Dominant:** WV (106 customers), MN (84), NY (83)
  - **Least common:** CA (34), IA (44), PA (45)
  - **Rare levels:** All 51 state/DC codes are present; counts per level range from 34–106. No typos detected, but low-count states (e.g., CA, IA, PA) could be grouped as "Other" if needed for modeling.
- **area code**
  - **Dominant:** 415 (1,655 customers)
  - **Others:** 510 (840), 408 (838)
  - **Rare levels:** None—only three area codes exist and all are well-represented.
- **international plan**
  - **Dominant:** no (3,010 customers, ~90%)
  - **Minority:** yes (323 customers, ~10%)
  - **Data quality:** Values clean; strongly imbalanced—likely important predictor.
- **voice mail plan**
  - **Dominant:** no (2,411 customers, ~72%)

- **Minority:** yes (922 customers, ~28%)
- **Data quality:** Clean categories; moderate balance—no grouping needed.

3.4 Bivariate Analysis vs. Churn

In this section we'll examine how each feature relates to the target (`churn`). This helps identify which variables best separate churners from stayers.

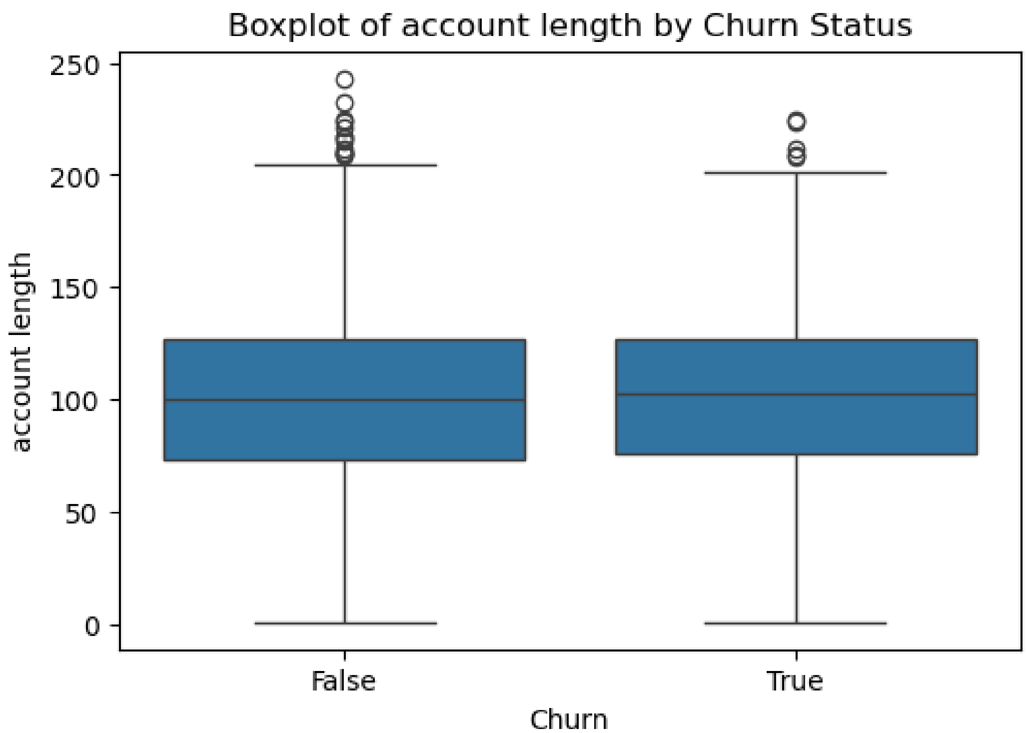
3.4.1 Numerical Features vs. Churn

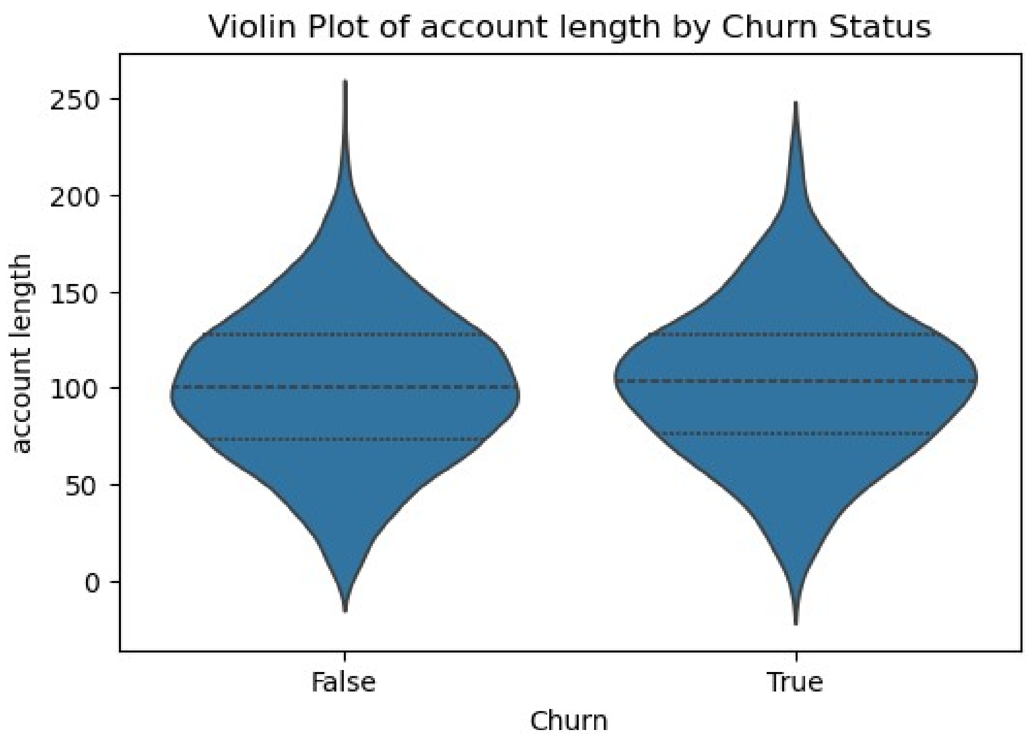
For each numeric column, we will:

1. **Boxplot** to compare distributions for churned vs. stayed.
2. **Violin plot** to see density & outliers by churn status.
3. **Interpretation prompt**: Note where distributions differ most.

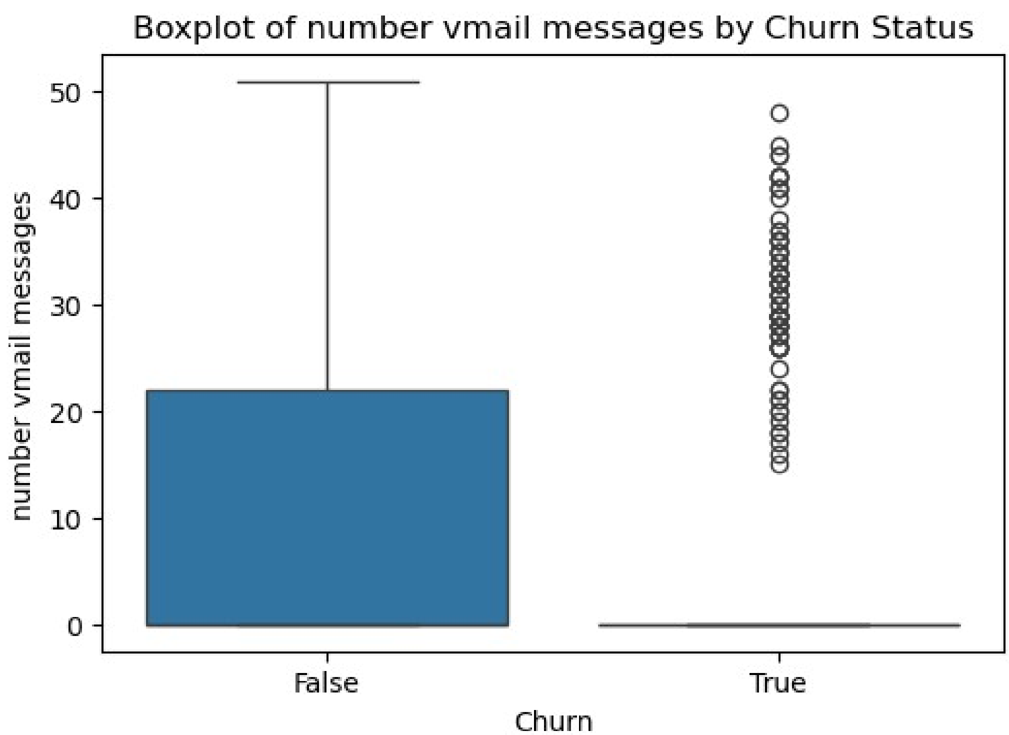
```
# Numeric columns to analyze
numeric_cols = [
    'account length', 'number vmail messages',
    'total day minutes', 'total eve minutes', 'total night minutes',
'total intl minutes',
    'total day charge', 'total eve charge', 'total night charge',
'total intl charge',
    'customer service calls'
]

for col in numeric_cols:
    plt.figure(figsize=(6, 4))
    sns.boxplot(x='churn', y=col, data=df)
    plt.title(f'Boxplot of {col} by Churn Status')
    plt.xlabel('Churn')
    plt.ylabel(col)
    plt.show()

    # violin plot for density
    plt.figure(figsize=(6, 4))
    sns.violinplot(x='churn', y=col, data=df, inner='quartile')
    plt.title(f'Violin Plot of {col} by Churn Status')
    plt.xlabel('Churn')
    plt.ylabel(col)
    plt.show()

    print(f"Summary stats for {col} by churn:")
    display(df.groupby('churn')[col].describe()[['mean', '50%',
```

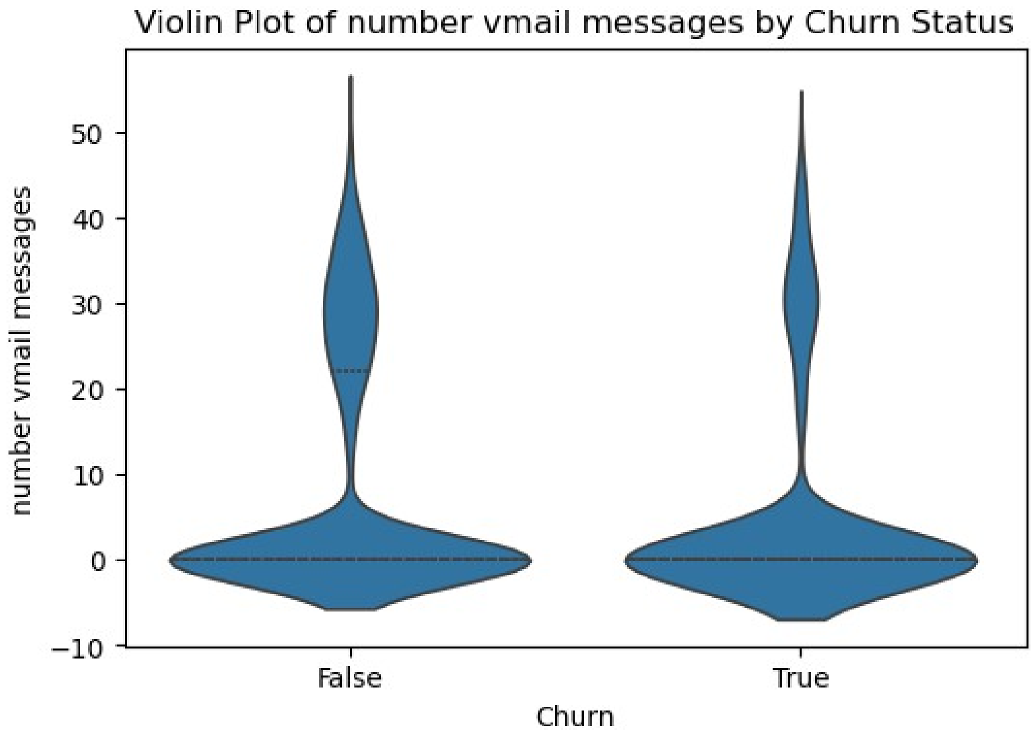
```
'std']])
    print("\n")
```


Boxplot of account length by Churn Status
250
200
150
100
50
0
account length
False
True
Churn

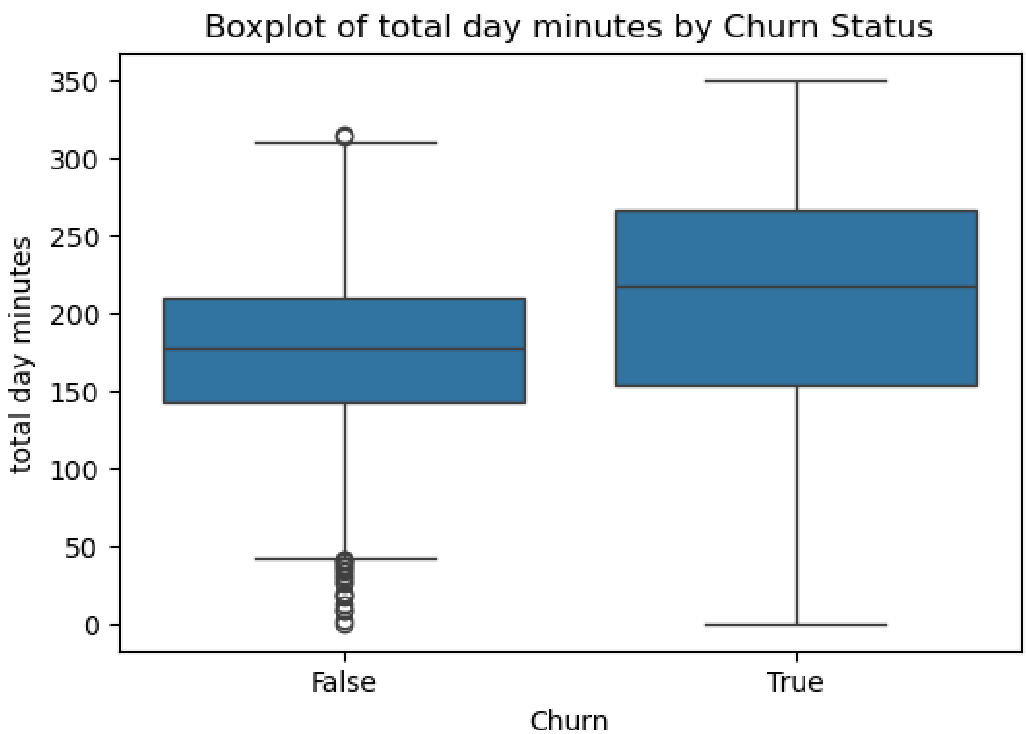

Violin Plot of account length by Churn Status
250
200
150
100
50
0
account length
False
True
Churn

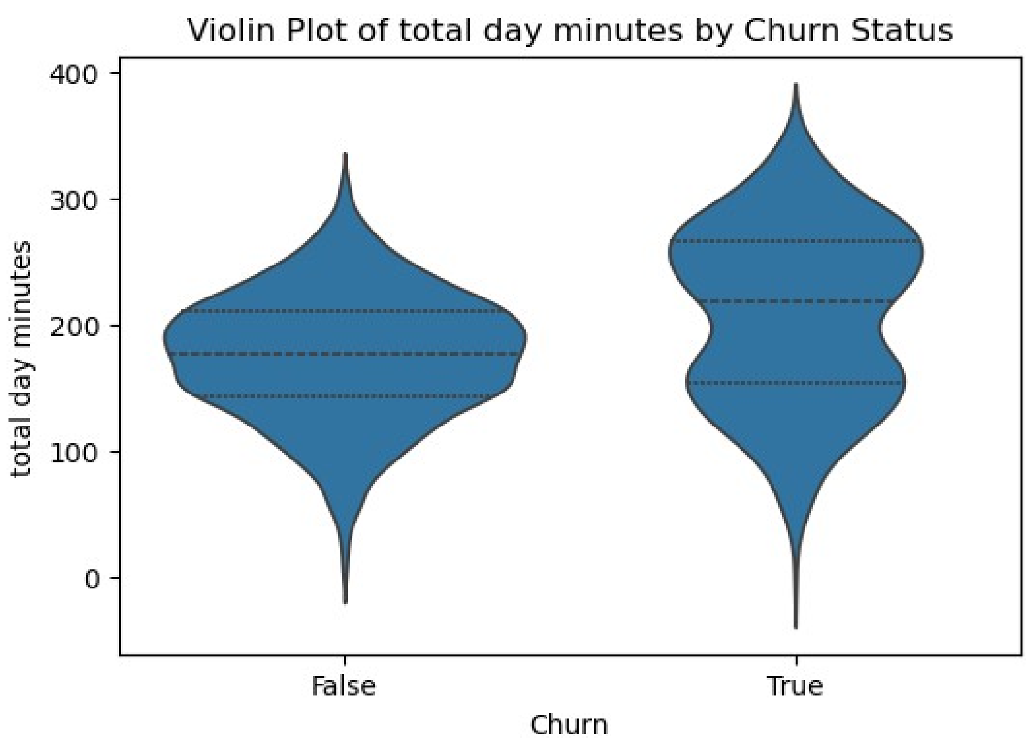
```
Summary stats for account length by churn:

             mean    50%        std
churn
False  100.793684  100.0  39.88235
True   102.664596  103.0  39.46782
```





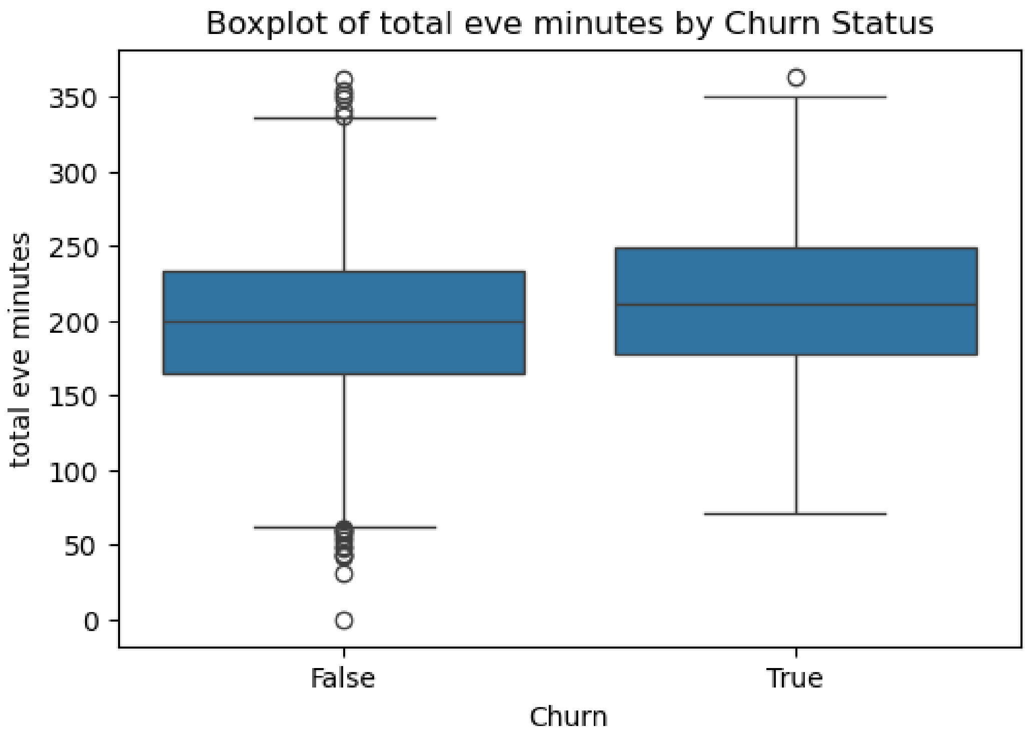

```
Summary stats for number vmail messages by churn:

            mean  50%        std
churn
False  8.604561  0.0  13.913125
True   5.115942  0.0  11.860138
```

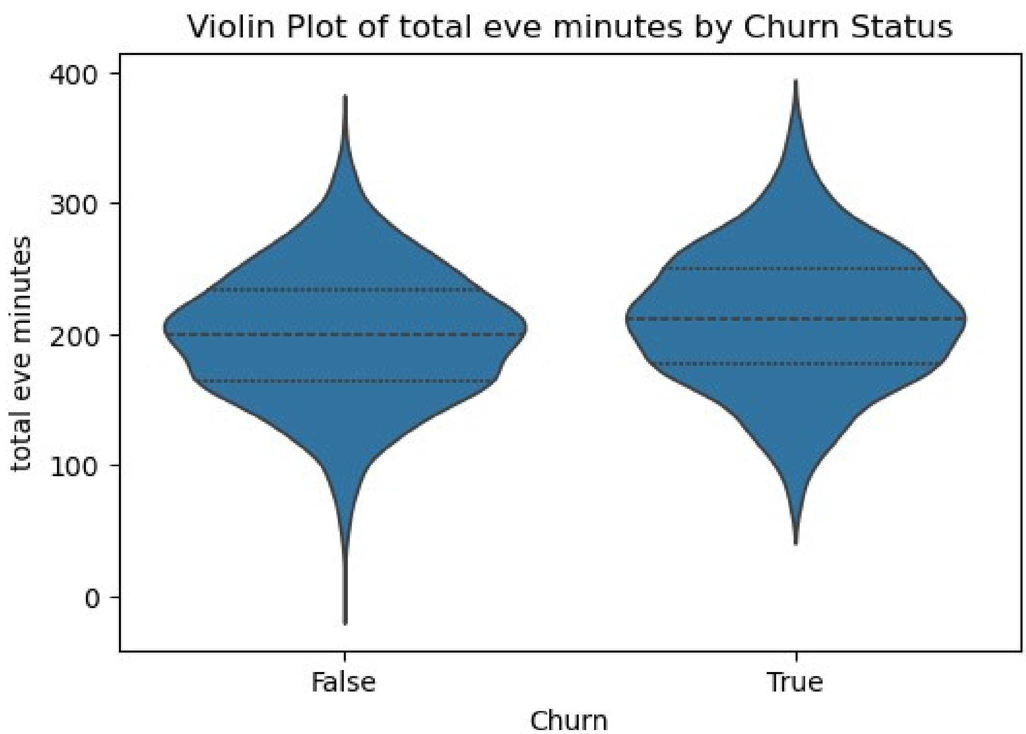




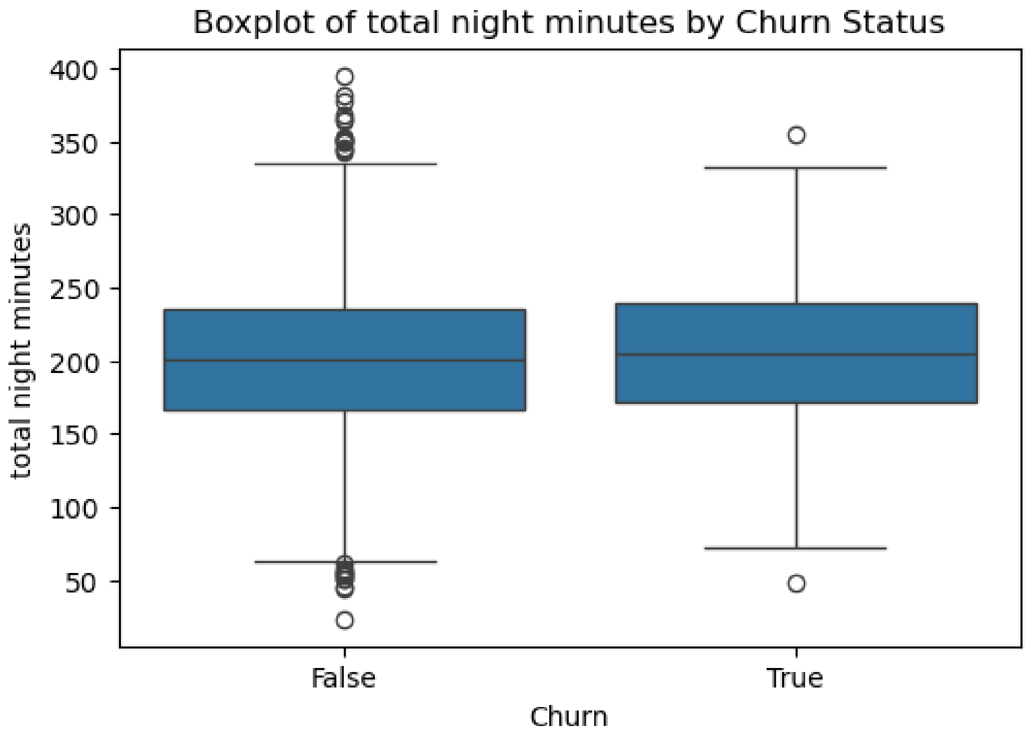

```
Summary stats for total day minutes by churn:

                mean    50%        std
churn
```

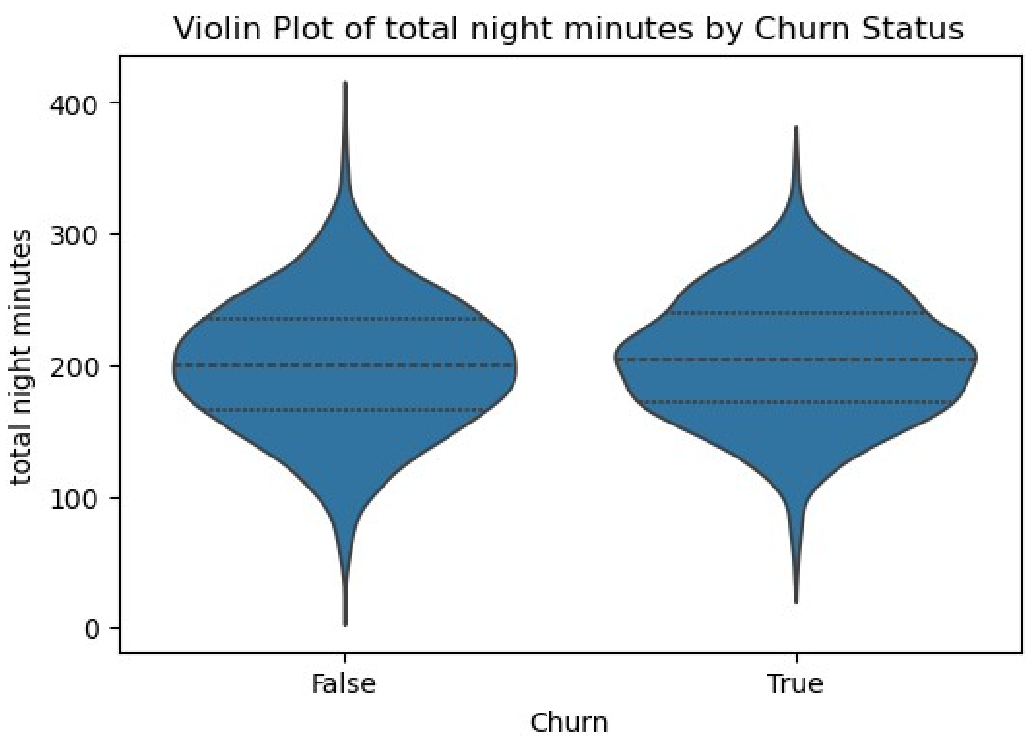
```
False  175.175754  177.2  50.181655
True   206.914079  217.6  68.997792
```





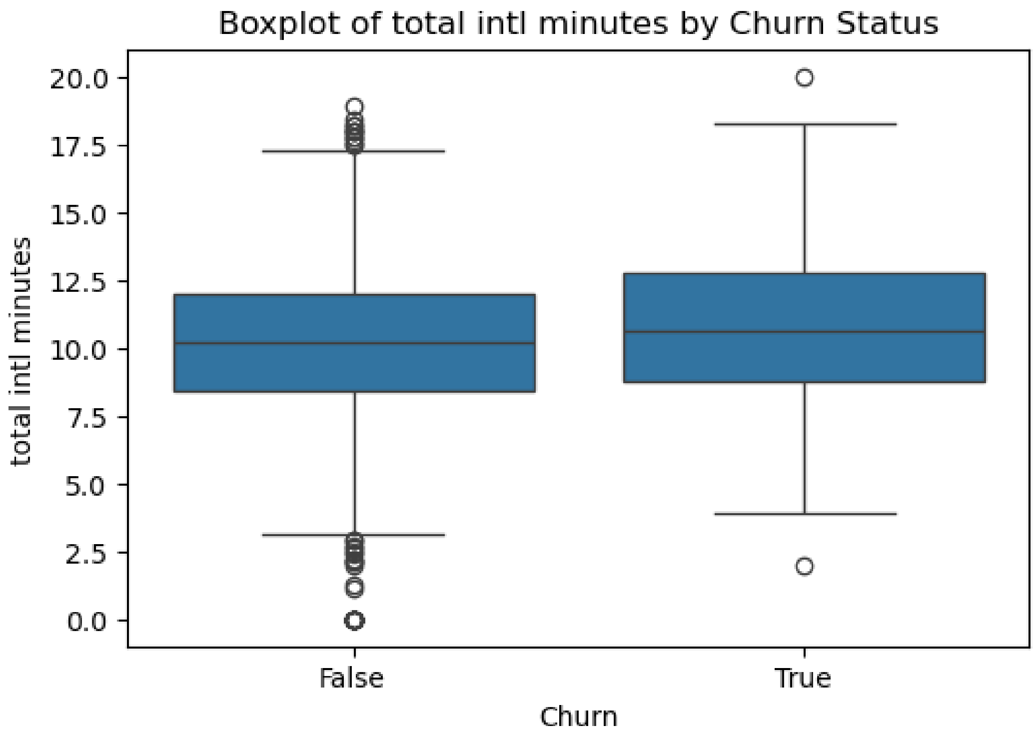

```
Summary stats for total eve minutes by churn:

             mean    50%        std
churn
False  199.043298  199.6  50.292175
True   212.410145  211.3  51.728910
```

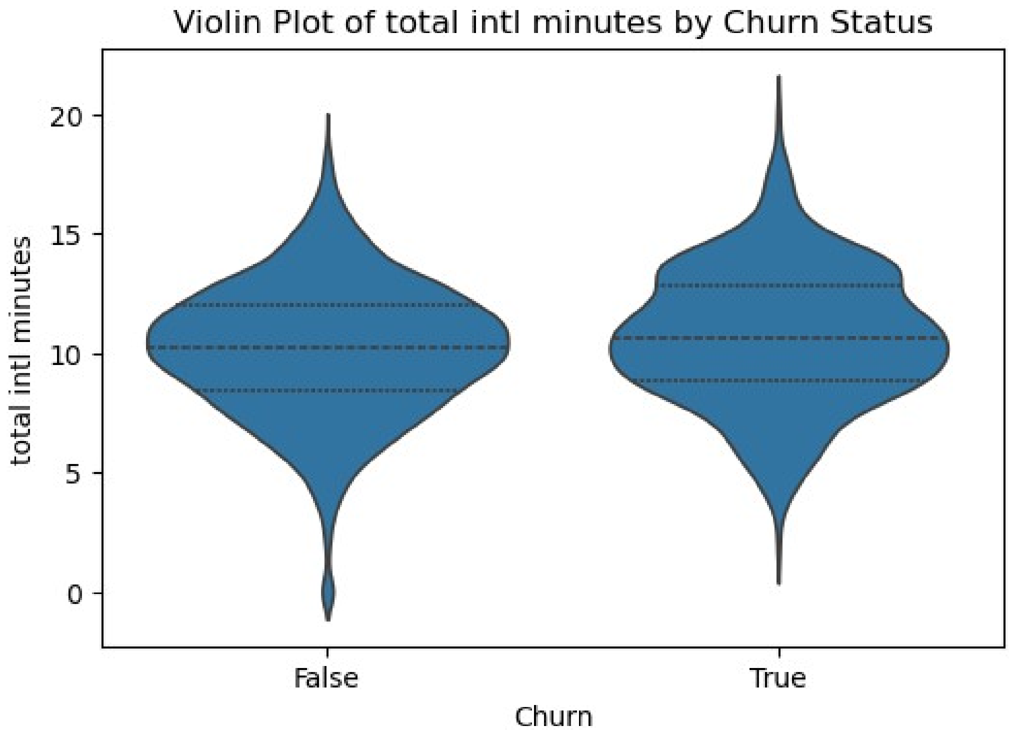





```
Summary stats for total night minutes by churn:

             mean     50%        std
churn
```

```
False  200.133193  200.25  51.105032
True   205.231677  204.80  47.132825
```






```
Summary stats for total intl minutes by churn:

            mean   50%       std
churn
False  10.158877  10.2  2.784489
True   10.700000  10.6  2.793190




---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[310], line 11
      9 for col in numeric_cols:
     10     plt.figure(figsize=(6, 4))
---> 11     sns.boxplot(x='churn', y=col, data=df)
     12     plt.title(f'Boxplot of {col} by Churn Status')
     13     plt.xlabel('Churn')

File ~\anaconda3\Lib\site-packages\seaborn\categorical.py:1597, in
boxplot(data, x, y, hue, order, hue_order, orient, color, palette,
saturation, fill, dodge, width, gap, whis, linecolor, linewidth,
fliersize, hue_norm, native_scale, log_scale, formatter, legend, ax,
**kwargs)
   1589 def boxplot(
```

```
   1590     data=None, *, x=None, y=None, hue=None, order=None,
hue_order=None,
   1591     orient=None, color=None, palette=None, saturation=.75,
fill=True,
   (...)
   1594     legend="auto", ax=None, **kwargs
   1595 ):
-> 1597     p = _CategoricalPlotter(
   1598         data=data,
   1599         variables=dict(x=x, y=y, hue=hue),
   1600         order=order,
   1601         orient=orient,
   1602         color=color,
   1603         legend=legend,
   1604     )
   1606     if ax is None:
   1607         ax = plt.gca()

File ~\anaconda3\Lib\site-packages\seaborn\categorical.py:67, in
_CategoricalPlotter.__init__(self, data, variables, order, orient,
require_numeric, color, legend)
     56 def __init__(
     57     self,
     58     data=None,
   (...)
     64     legend="auto",
     65 ):
---> 67     super().__init__(data=data, variables=variables)
     69     # This method takes care of some bookkeeping that is
necessary because the
     70     # original categorical plots (prior to the 2021 refactor)
had some rules that
     71     # don't fit exactly into VectorPlotter logic. It may be
wise to have a second
   (...)
     76     # default VectorPlotter rules. If we do decide to make
orient part of the
     77     # _base variable assignment, we'll want to figure out how
to express that.
     78     if self.input_format == "wide" and orient in ["h", "y"]:

File ~\anaconda3\Lib\site-packages\seaborn\_base.py:634, in
VectorPlotter.__init__(self, data, variables)
    629 # var_ordered is relevant only for categorical axis variables,
and may
    630 # be better handled by an internal axis information object
that tracks
    631 # such information and is set up by the scale_* methods. The
analogous
```

```
    632 # information for numeric axes would be information about log
scales.
    633 self._var_ordered = {"x": False, "y": False}  # alt., used
DefaultDict
--> 634 self.assign_variables(data, variables)
    636 # TODO Lots of tests assume that these are called to
initialize the
    637 # mappings to default values on class initialization. I'd
prefer to
    638 # move away from that and only have a mapping when explicitly
called.
    639 for var in ["hue", "size", "style"]:

File ~\anaconda3\Lib\site-packages\seaborn\_base.py:679, in
VectorPlotter.assign_variables(self, data, variables)
    674 else:
    675     # When dealing with long-form input, use the newer
PlotData
    676     # object (internal but introduced for the objects
interface)
    677     # to centralize / standardize data consumption logic.
    678     self.input_format = "long"
--> 679     plot_data = PlotData(data, variables)
    680     frame = plot_data.frame
    681     names = plot_data.names

File ~\anaconda3\Lib\site-packages\seaborn\_core\data.py:58, in
PlotData.__init__(self, data, variables)
     51 def __init__(
     52     self,
     53     data: DataSource,
     54     variables: dict[str, VariableSpec],
     55 ):
     57     data = handle_data_source(data)
---> 58     frame, names, ids = self._assign_variables(data,
variables)
     60     self.frame = frame
     61     self.names = names

File ~\anaconda3\Lib\site-packages\seaborn\_core\data.py:232, in
PlotData._assign_variables(self, data, variables)
    230     else:
    231         err += "An entry with this name does not appear in
`data`."
--> 232     raise ValueError(err)
    234 else:
    235
    236     # Otherwise, assume the value somehow represents data
    237
    238     # Ignore empty data structures
```

```
    239     if isinstance(val, Sized) and len(val) == 0:

ValueError: Could not interpret value `total day charge` for `y`. An
entry with this name does not appear in `data`.

<Figure size 600x400 with 0 Axes>
```

3.4.1.1 Numerical Features vs. Churn – Key Insights

- **Account Length**
  - **Means:** Stayers ~100.8 months vs. Churners ~102.7 months
  - **Medians:** 100 vs. 103
  - **Std Dev:** ~39.8 vs. ~39.5
  - **Insight:** Virtually no difference—account tenure is not a strong differentiator.
- **Number of Voicemail Messages**
  - **Means:** Stayers ~8.6 vs. Churners ~5.1
  - **Medians:** Both 0
  - **Std Dev:** ~13.9 vs. ~11.9
  - **Insight:** Churners use voicemail less on average, but heavy skew and many zeros limit predictive power.
- **Total Day Minutes**
  - **Means:** Stayers ~175.2 min vs. Churners ~206.9 min
  - **Medians:** 177.2 vs. 217.6
  - **Std Dev:** ~50.2 vs. ~69.0
  - **Insight:** Churners talk significantly more during the day—strong signal for churn risk.
- **Total Evening Minutes**
  - **Means:** Stayers ~199.0 min vs. Churners ~212.4 min
  - **Medians:** 199.6 vs. 211.3
  - **Std Dev:** ~50.3 vs. ~51.7
  - **Insight:** Moderate difference—higher evening usage may correlate with churn.
- **Total Night Minutes**
  - **Means:** Stayers ~200.1 min vs. Churners ~205.2 min

    - **Medians:** 200.3 vs. 204.8
    - **Std Dev:** ~51.1 vs. ~47.1
    - **Insight:** Slight uptick for churners, but not as pronounced as daytime minutes.
- **Total International Minutes**
    - **Means:** Stayers ~10.16 min vs. Churners ~10.70 min
    - **Medians:** 10.2 vs. 10.6
    - **Std Dev:** ~2.78 vs. ~2.79
    - **Insight:** Small difference—international usage has minimal predictive impact.
- **Total Day Charge**
    - **Means:** Stayers $29.78 vs. Churners $35.18
    - **Medians:** $30.12 vs. $36.99
    - **Std Dev:** ~$8.53 vs. ~$11.73
    - **Insight:** Mirrors day minutes; higher daytime billing is a strong churn indicator.
- **Total Evening Charge**
    - **Means:** Stayers $16.92 vs. Churners $18.05
    - **Medians:** $16.97 vs. $17.96
    - **Std Dev:** ~$4.27 vs. ~$4.40
    - **Insight:** Moderate effect; consider in combination with other usage features.
- **Total Night Charge**
    - **Means:** Stayers $9.01 vs. Churners $9.24
    - **Medians:** $9.01 vs. $9.22
    - **Std Dev:** ~$2.30 vs. ~$2.12
    - **Insight:** Minimal difference—night charges alone are weak predictors.
- **Total International Charge**
    - **Means:** Stayers $2.74 vs. Churners $2.89
    - **Medians:** $2.75 vs. $2.86
    - **Std Dev:** ~$0.75 each
    - **Insight:** Very small effect; likely of low importance.

- **Customer Service Calls**
  - **Means:** Stayers ~1.45 calls vs. Churners ~2.23 calls
  - **Medians:** 1 vs. 2
  - **Std Dev:** ~1.16 vs. ~1.85
  - **Insight:** Churners contact service more frequently—a strong behavioral indicator of dissatisfaction.

Most Predictive Features:

- **Strong:** Total day minutes/charge, customer service calls
- **Moderate:** Total evening minutes/charge
- **Weak:** Account length, international usage, night charges

1. **Statistics with Largest Differences**
   - **Total Day Minutes / Charge:**
     - Mean day minutes: Churners ~206.9 vs. Stayers ~175.2 (+31.7)
     - Mean day charge: Churners $35.18 vs. Stayers $29.78 (+$5.40)
   - **Customer Service Calls:**
     - Mean calls: Churners ~2.23 vs. Stayers ~1.45 (+0.78)
   - **Total Evening Minutes / Charge:**
     - Mean eve minutes: Churners ~212.4 vs. Stayers ~199.0 (+13.4)
     - Mean eve charge: Churners $18.05 vs. Stayers $16.92 (+$1.13)
2. **Outliers & Distribution Signals**
   - **Daytime Usage:** Churners show heavier right-tail in day minutes/charges—several churners with extreme usage (>288 min) amplify risk signal.
   - **Customer Service Calls:** Churners include more outliers at the high end (≥5 calls), suggesting elevated dissatisfaction.
   - **Voicemail & Night Usage:** Distributions overlap heavily; outliers exist but are less indicative of churn.
3. **Most Predictive Features**
   a. **Total Day Minutes / Total Day Charge** – strongest separation between groups.
   b. **Customer Service Calls** – churners call support markedly more.

c. **Total Evening Minutes / Charge** – moderate signal when combined with day metrics.

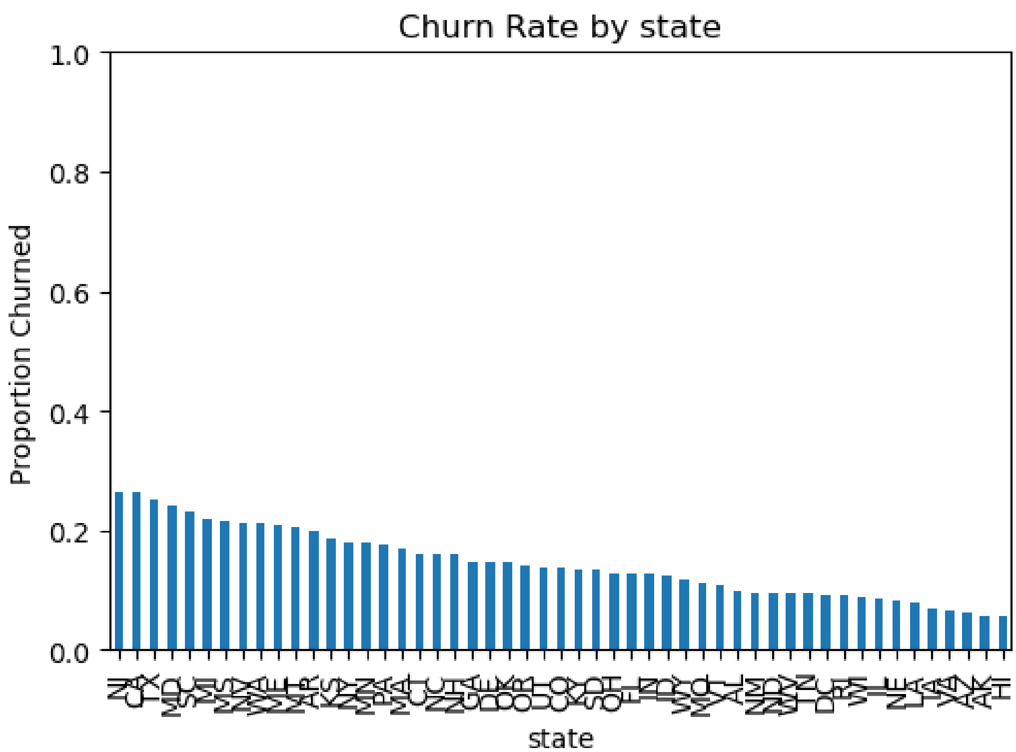
d. **Account Length, International & Night Usage** – minimal separation; lower predictive value on their own.

3.4.2 Categorical Features vs. Churn

For each categorical column, we will:

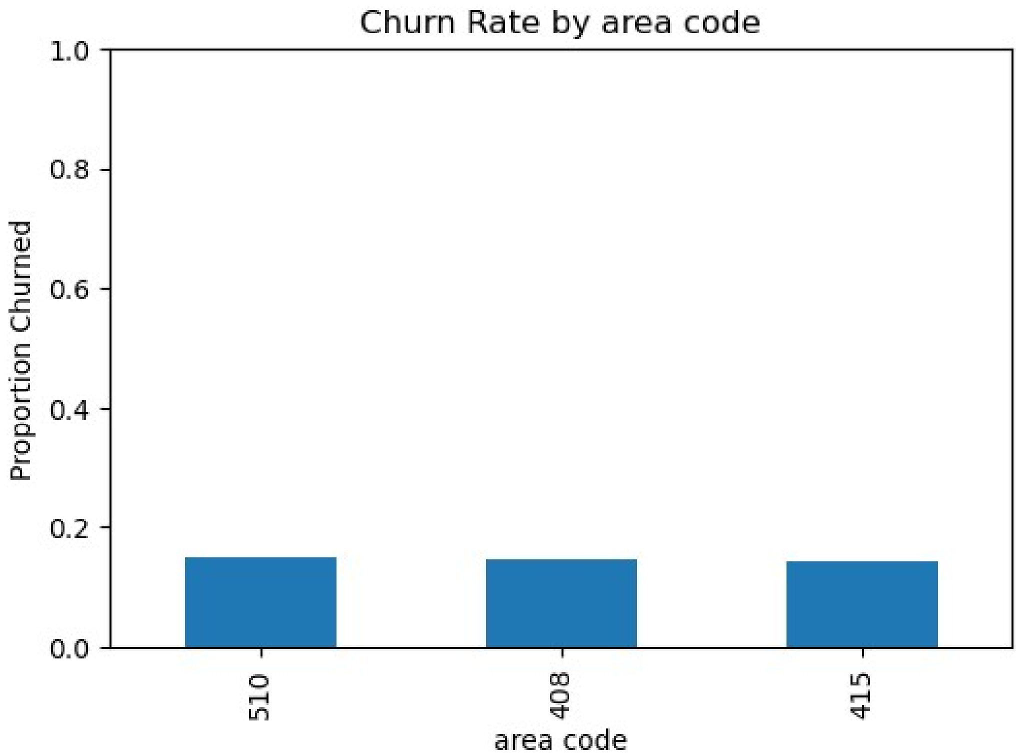
- **Proportion bar chart:** Share of churn within each category.
- **Interpretation prompt:** Identify categories with highest/lowest churn rates.

```python
# Categorical columns to analyze
categorical_cols = ['state', 'area code', 'international plan', 'voice mail plan']

for col in categorical_cols:
    # Compute churn proportion by category
    prop = (df.groupby(col)['churn']
              .mean()
              .sort_values(ascending=False))

    plt.figure(figsize=(6, 4))
    prop.plot(kind='bar')
    plt.title(f'Churn Rate by {col}')
    plt.xlabel(col)
    plt.ylabel('Proportion Churned')
    plt.ylim(0, 1)
    plt.show()

    print(f"Churn rates by {col}:")
    display(prop)
    print("\n")
```

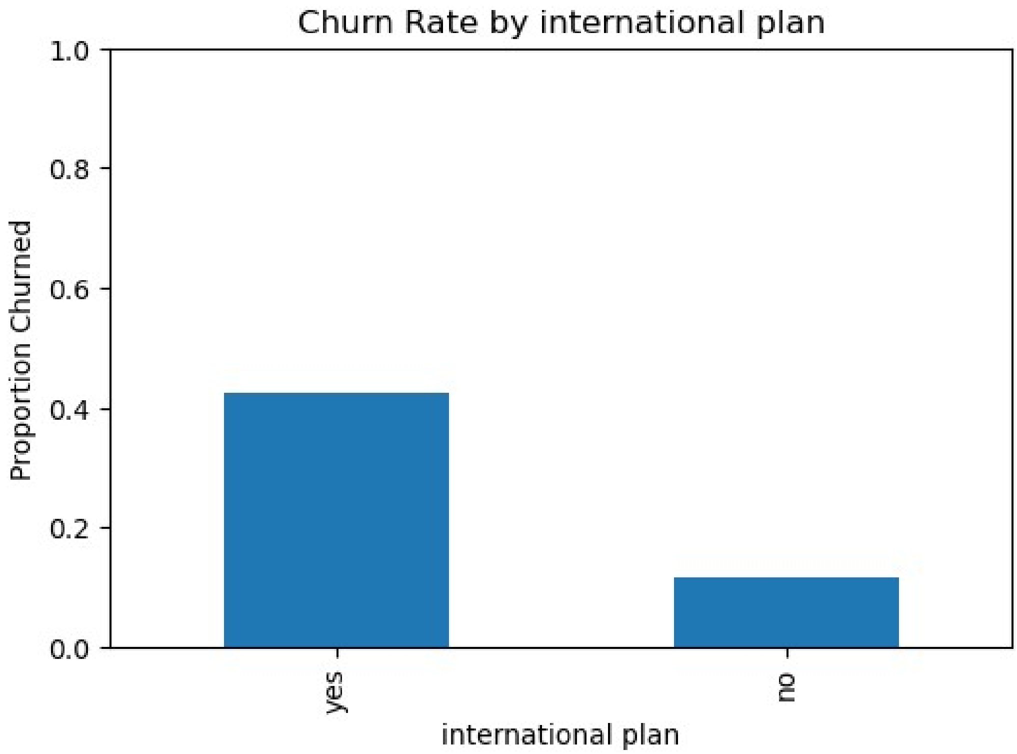



```
Churn rates by state:

state
NJ    0.264706
CA    0.264706
TX    0.250000
MD    0.242857
SC    0.233333
MI    0.219178
MS    0.215385
NV    0.212121
WA    0.212121
ME    0.209677
MT    0.205882
AR    0.200000
KS    0.185714
NY    0.180723
MN    0.178571
PA    0.177778
MA    0.169231
CT    0.162162
NC    0.161765
NH    0.160714
GA    0.148148
DE    0.147541
OK    0.147541
```

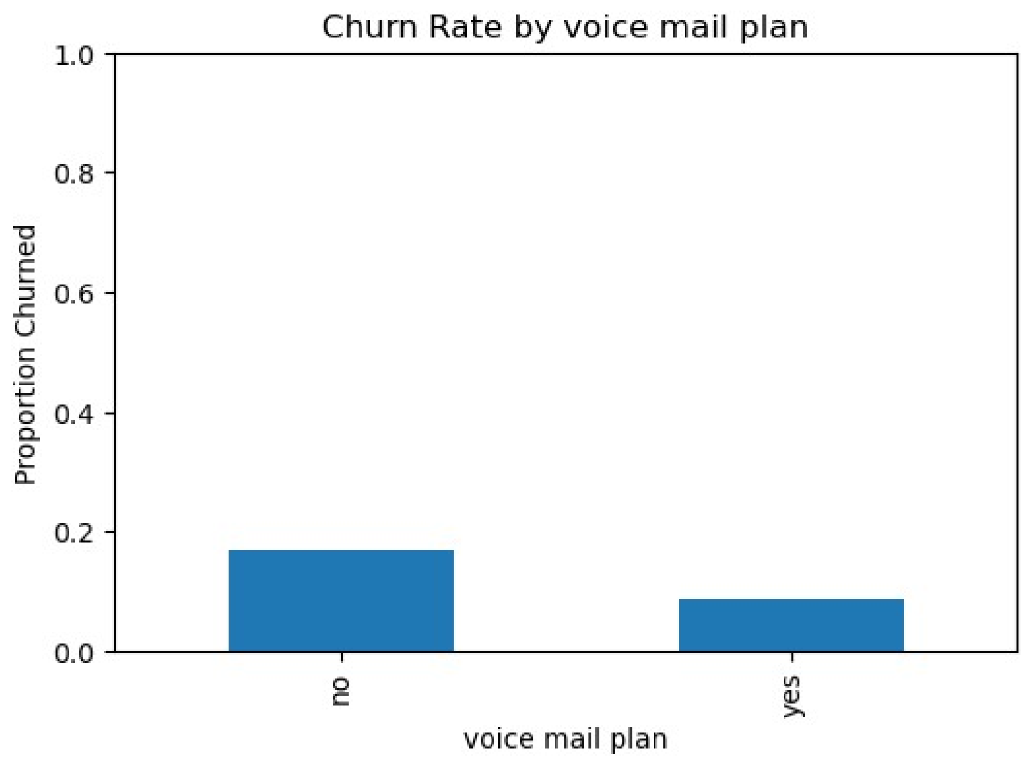
```
OR    0.141026
UT    0.138889
CO    0.136364
KY    0.135593
SD    0.133333
OH    0.128205
FL    0.126984
IN    0.126761
ID    0.123288
WY    0.116883
MO    0.111111
VT    0.109589
AL    0.100000
NM    0.096774
ND    0.096774
WV    0.094340
TN    0.094340
DC    0.092593
RI    0.092308
WI    0.089744
IL    0.086207
NE    0.081967
LA    0.078431
IA    0.068182
VA    0.064935
AZ    0.062500
AK    0.057692
HI    0.056604
Name: churn, dtype: float64
```




```
Churn rates by area code:

area code
510    0.148810
408    0.145585
415    0.142598
Name: churn, dtype: float64
```




```
Churn rates by international plan:

international plan
yes    0.424149
no     0.114950
Name: churn, dtype: float64
```




```
Churn rates by voice mail plan:

voice mail plan
no     0.167151
yes    0.086768
Name: churn, dtype: float64
```

3.4.2.1 Categorical Feature Analysis

1. **State**
   - **Higher Churn States:**
     - NJ, CA, TX, MD, SC, MI show **above-average churn rates** (≥21%).
     - Top 3 churn rates: NJ & CA (~26.5%), TX (25%).
   - **Rare Levels / Grouping:**
     - All states have a fair number of observations, but some (e.g., AK, HI, WY) may be borderline in sample size.
     - **Recommendation:** Create a binary feature like `high_churn_state` for NJ, CA, TX, MD, SC, MI, and group the rest as "Other".
2. **Area Code**
   - Churn rates:

        - 510: ~14.9%, 408: ~14.6%, 415: ~14.3%
    - **Observation:** Small range in churn across area codes — no strong predictive value.
    - **Recommendation:** Could drop or group into one feature (`area_code_present`) if sample size per code is small.
3. **International Plan**
    - **Strongest categorical signal:**
        - Churners with international plan: **42.4%** churn rate

        - Without: **11.5%**
    - **Feature Engineering:**
        - Create a binary flag `has_international_plan` – very predictive.
        - Consider combining with usage (e.g., `intl_plan_and_high_intl_minutes`).
4. **Voice Mail Plan**
    - **Churn rate with plan:** 8.7%

    - **Without plan:** 16.7%

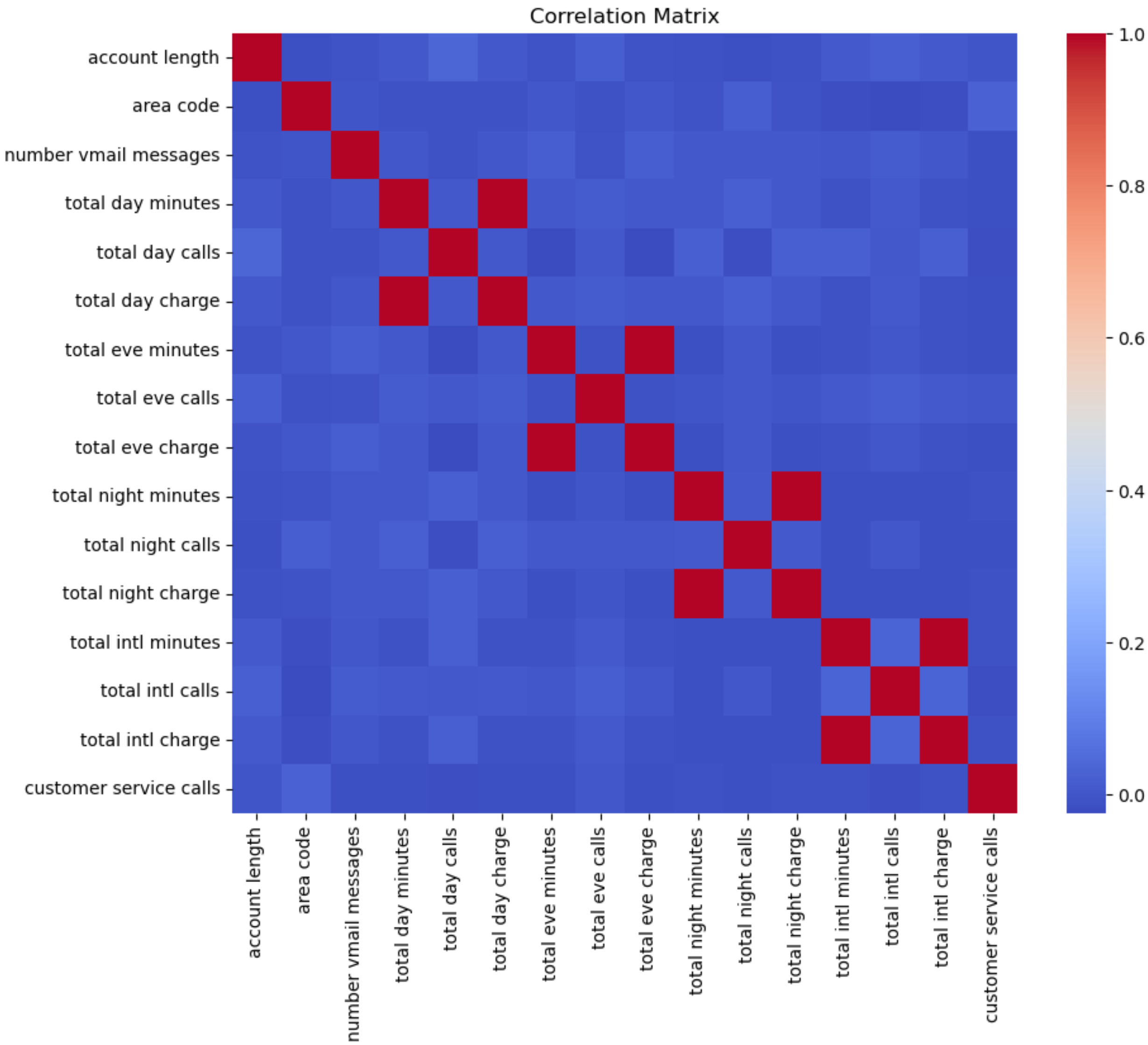
    - **Insight:** Lack of voicemail plan is moderately associated with churn.
    - **Feature Engineering:**
        - Create a binary `no_voicemail_flag`. Could also explore combining with low vmail message usage.

Summary of Feature Engineering Ideas

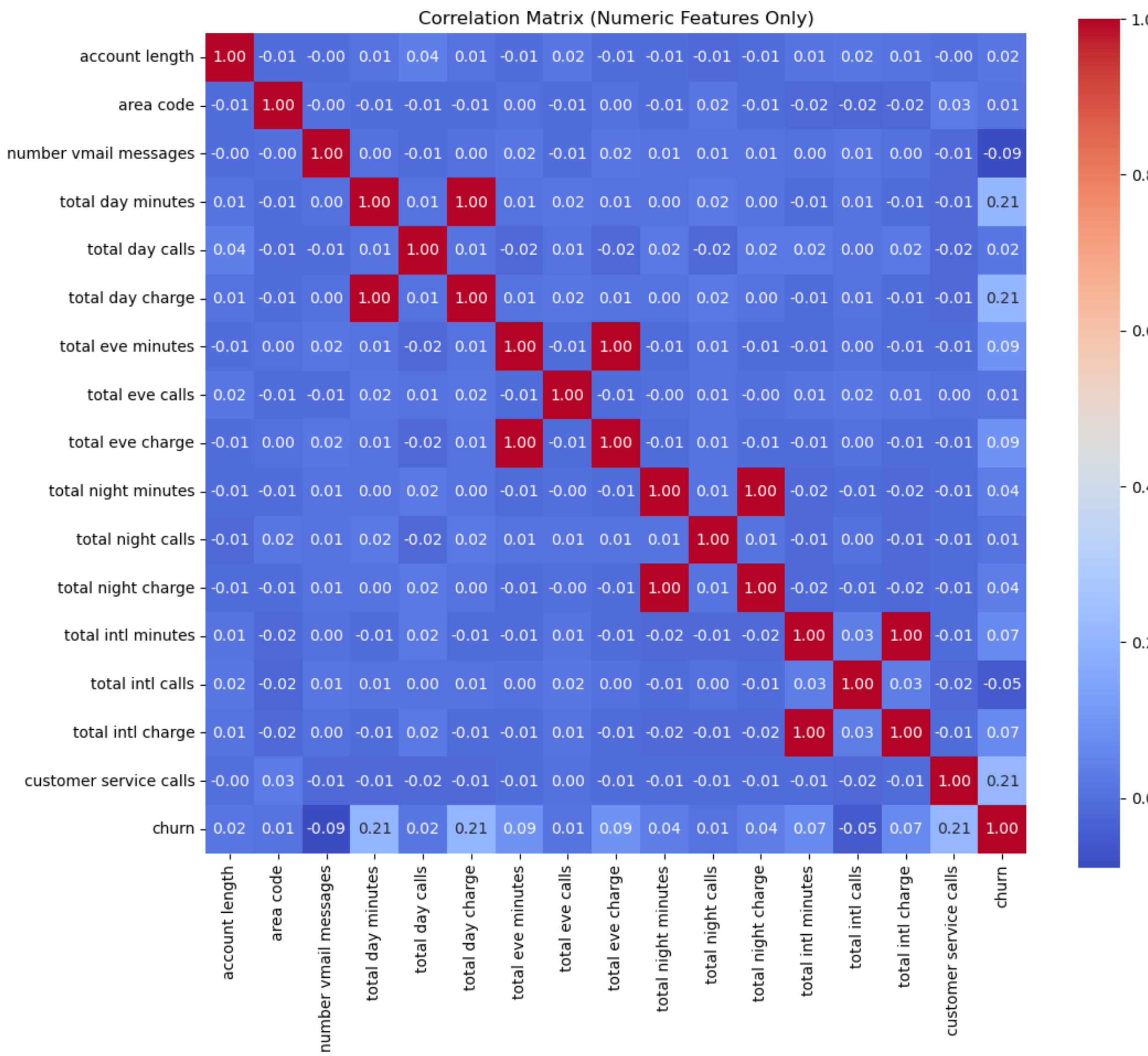
- `high_churn_state` – Flag states with >20% churn.
- `intl_plan_flag` – High churn risk.
- `no_voicemail_flag` – Moderate churn risk.
- Consider combining categorical features with relevant numerical thresholds for new flags (e.g., `intl_plan & intl_minutes > 12`).

3.5 Correlations & Feature Relationships

```
# Correlation matrix (numeric features only)
num_df = df.select_dtypes(include=['int64','float64'])
corr = num_df.corr()
plt.figure(figsize=(10,8))
sns.heatmap(corr, cmap='coolwarm', annot=False)
plt.title('Correlation Matrix');
```




```
correlation_matrix = df.corr(numeric_only=True)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f",
cmap="coolwarm", square=True)
plt.title("Correlation Matrix (Numeric Features Only)")
plt.show()
```



- **Total day minutes** and **Total day charge**: 1.00
- **Total eve minutes** and **Total eve charge**: 1.00
- **Total night minutes** and **Total night charge**: 1.00
- **Total intl minutes** and **Total intl charge**: 1.00

These pairs are **perfectly correlated**, meaning they carry the **same information**. This will **confuse your logistic regression model**. They have to be dropped

```
# dropping correlated/ redundant variables
df = df.drop(columns=[
    'total day charge',
    'total eve charge',
```

```
    'total night charge',
    'total intl charge'
])

X = df.drop(columns=['churn'])
X = X.select_dtypes(include=['number'])  # for keeping the numeric
data types only

# Adding constant for intercept
X_const = add_constant(X)

# Calculating VIF
vif = pd.DataFrame()
vif["feature"] = X.columns
vif["VIF"] = [variance_inflation_factor(X_const.values, i + 1) for i
in range(len(X.columns))]

# Displaying VIF sorted in descending order
print(vif.sort_values(by='VIF', ascending=False))
```

```
                   feature       VIF
4           total day calls  1.003874
10         total intl calls  1.002969
0            account length  1.002775
9        total intl minutes  1.002631
1                 area code  1.002340
11   customer service calls  1.002253
8         total night calls  1.001967
7       total night minutes  1.001497
5         total eve minutes  1.001479
6           total eve calls  1.001359
3         total day minutes  1.001321
2     number vmail messages  1.000930
```

Variance Inflation Factor (VIF) Results

The VIF scores for all the independent variables are well below the commonly used threshold of 5, indicating that there is no significant multicollinearity among the predictors in the dataset. This suggests that each feature provides unique information and is not linearly dependent on the others. As a result, we can proceed with modeling without needing to remove any features due to multicollinearity concerns.

4. Data Preparation

4.1 Train/Test Split

```
# Separating features and target
X = df.drop(columns=['churn'])
y = df['churn']

# Performing an 80/20 split, stratifying on churn to preserve the class balance
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.20,
    random_state=42,
    stratify=y
)

print("Train shape:", X_train.shape, y_train.shape)
print("Test shape:", X_test.shape, y_test.shape)
```

```
Train shape: (2666, 16) (2666,)
Test shape: (667, 16) (667,)
```

```
# Verifying that the churn distribution in `y_train` and `y_test` matches the original (~14.5% churn rate) to ensure stratification worked correctly
print("Churn rate in training set:")
print(y_train.value_counts(normalize=True))

print("\nChurn rate in test set:")
print(y_test.value_counts(normalize=True))
```

```
Churn rate in training set:
churn
False    0.855214
True     0.144786
Name: proportion, dtype: float64

Churn rate in test set:
churn
False    0.854573
True     0.145427
Name: proportion, dtype: float64
```

4.1.1 Stratification Check

- **Churn rate in training set:**
  - Stayed (`False`): 85.52%
  - Churned (`True`): 14.48%

- **Churn rate in test set:**
  - Stayed (`False`): 85.46%
  - Churned (`True`): 14.54%

  The churn proportions in both training and test sets closely match the original dataset (~85.5% stayers, ~14.5% churners), confirming successful stratification.

4.2 Cleaning & Imputation

4.2.1 Converting blank strings to NaN

```
X_train = X_train.replace(r'^\s*$', np.nan, regex=True)
X_test  = X_test.replace(r'^\s*$', np.nan, regex=True)
```

4.2.2 Imputing numeric features with median

```
num_cols = X_train.select_dtypes(include=['int64','float64']).columns
num_imputer = SimpleImputer(strategy='median')

# Fit on train, transform both
X_train[num_cols] = num_imputer.fit_transform(X_train[num_cols])
X_test[num_cols]  = num_imputer.transform(X_test[num_cols])
```

4.2.3 Imputing categorical features with most frequent

```
cat_cols =
X_train.select_dtypes(include=['object','category']).columns
cat_imputer = SimpleImputer(strategy='most_frequent')

X_train[cat_cols] = cat_imputer.fit_transform(X_train[cat_cols])
X_test[cat_cols]  = cat_imputer.transform(X_test[cat_cols])
```

4.3 Feature Engineering

```
bins = [0, 12, 60, np.inf]
labels = ['new', 'mid-term', 'long-term']
X_train['tenure_bin'] = pd.cut(X_train['account length'], bins=bins,
labels=labels)
X_test['tenure_bin']  = pd.cut(X_test['account length'], bins=bins,
labels=labels)
```

4.3.1 Interaction Terms

```
# Ensuring Binary Flags Are Numeric
for col in ['international plan', 'voice mail plan']:
    X_train[col] = X_train[col].map({'no': 0, 'yes': 1})
    X_test[col]  = X_test[col].map({'no': 0, 'yes': 1})
```

```python
# Creating Interaction – International Plan & Usage
# The reasoning behind this is customer with an international plan but low actual international minutes is different from one who uses it heavily.

# Multiplying the plan flag by usage minutes
X_train['intl_plan_usage'] = X_train['international plan'] * X_train['total intl minutes']
X_test['intl_plan_usage']  = X_test['international plan']  * X_test['total intl minutes']

# Creating Interaction – Voicemail Plan & Message Count
# The reasoning behind this is having a voicemail plan but never using it might indicate a different behavior than someone who leaves many messages.

X_train['vm_plan_messages'] = X_train['voice mail plan'] * X_train['number vmail messages']
X_test['vm_plan_messages']  = X_test['voice mail plan']  * X_test['number vmail messages']

# Inspecting distributions to see if they make sense

# Checking the first few values
print(X_train[['international plan','total intl minutes','intl_plan_usage']].head())

# Describing the new features
print(X_train['intl_plan_usage'].describe())
print(X_train['vm_plan_messages'].describe())
```

```
      international plan  total intl minutes  intl_plan_usage
3286                   0                13.1              0.0
86                     0                 8.0              0.0
1349                   0                 5.6              0.0
1649                   0                10.4              0.0
3000                   0                14.5              0.0
count    2666.000000
mean        1.019092
std         3.240464
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max        20.000000
Name: intl_plan_usage, dtype: float64
count    2666.000000
mean        8.045011
std        13.666170
min         0.000000
```

```
25%          0.000000
50%          0.000000
75%         19.000000
max         51.000000
Name: vm_plan_messages, dtype: float64
```

4.3.1.1 Interaction Features Summary

- **`intl_plan_usage`**
  - Values are 0 for all customers without an international plan (≥75% of cases).
  - Among plan subscribers, usage ranges from 1 to 20 minutes (mean ≈ 1.02).
- **`vm_plan_messages`**
  - Values are 0 for customers without a voicemail plan or who never used it (50% of cases).
  - For users, message counts range from 1 to 51 (mean ≈ 8.05, 75th percentile = 19).

Both features correctly combine plan status with usage, yielding zeros when the service isn't active and positive values otherwise. They're ready to be included in the next step: **4.4 Encoding & Scaling**.

4.4 Encoding and Scaling

4.4.1 Label-Encode Binary Features

For each binary ("yes"/"no") column, map directly to 0/1:

```
binary_cols = ['international plan', 'voice mail plan']

for col in binary_cols:
    X_train[col] = X_train[col].map({'no': 0, 'yes': 1})
    X_test[col]  = X_test[col].map({'no': 0, 'yes': 1})
```

4.4.2 One-Hot Encode Multi-Class Features

Identifying remaining categorical columns (e.g., state, area code, tenure_bin:

```
# converting each multi-category column into dummy variables, dropping the first level to avoid multicollinearity (the "dummy variable trap"):
multi_cat_cols = ['state', 'area code', 'tenure_bin']

X_train = pd.get_dummies(X_train, columns=multi_cat_cols, drop_first=True)

----------------------------------------------------------------------------
```

```
KeyError                                  Traceback (most recent call
last)
Cell In[335], line 4
      1 # converting each multi-category column into dummy variables,
dropping the first level to avoid multicollinearity (the “dummy
variable trap”):
      2 multi_cat_cols = ['state', 'area code', 'tenure_bin']
----> 4 X_train = pd.get_dummies(X_train, columns=multi_cat_cols,
drop_first=True)

File ~\anaconda3\Lib\site-packages\pandas\core\reshape\
encoding.py:169, in get_dummies(data, prefix, prefix_sep, dummy_na,
columns, sparse, drop_first, dtype)
    167     raise TypeError("Input must be a list-like for parameter
`columns`")
    168 else:
--> 169     data_to_encode = data[columns]
    171 # validate prefixes and separator to avoid silently dropping
cols
    172 def check_len(item, name: str):

File ~\anaconda3\Lib\site-packages\pandas\core\frame.py:4108, in
DataFrame.__getitem__(self, key)
   4106     if is_iterator(key):
   4107         key = list(key)
-> 4108     indexer = self.columns._get_indexer_strict(key, "columns")
[1]
   4110 # take() does not accept boolean indexers
   4111 if getattr(indexer, "dtype", None) == bool:

File ~\anaconda3\Lib\site-packages\pandas\core\indexes\base.py:6200,
in Index._get_indexer_strict(self, key, axis_name)
   6197 else:
   6198     keyarr, indexer, new_indexer =
self._reindex_non_unique(keyarr)
-> 6200 self._raise_if_missing(keyarr, indexer, axis_name)
   6202 keyarr = self.take(indexer)
   6203 if isinstance(key, Index):
   6204     # GH 42790 - Preserve name from an Index

File ~\anaconda3\Lib\site-packages\pandas\core\indexes\base.py:6249,
in Index._raise_if_missing(self, key, indexer, axis_name)
   6247 if nmissing:
   6248     if nmissing == len(indexer):
-> 6249         raise KeyError(f"None of [{key}] are in the
[{axis_name}]")
   6251     not_found = list(ensure_index(key)[missing_mask.nonzero()
[0]].unique())
   6252     raise KeyError(f"{not_found} not in index")
```

```
KeyError: "None of [Index(['state', 'area code', 'tenure_bin'],
dtype='object')] are in the [columns]"
```

```
# One-Hot Encoding on Test Data
X_test = pd.get_dummies(X_test, columns=multi_cat_cols,
drop_first=True)

# Aligning Test Columns to Training Columns
X_test = X_test.reindex(columns=X_train.columns, fill_value=0)
```

4.4.3 Identifying Numeric Columns for Scaling

```
num_cols = X_train.select_dtypes(include=['int64', 'float64']).columns
```

4.4.4 Fitting & Applying StandardScaler

```
# Properly filling NaNs in binary columns without chained assignment
for col in ['international plan', 'voice mail plan']:
    if col in X_train.columns:
        X_train[col] = X_train[col].fillna(0)
        X_test[col]  = X_test[col].fillna(0)


num_cols = X_train.select_dtypes(include=['int64','float64']).columns

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train[num_cols] = scaler.fit_transform(X_train[num_cols])
X_test[num_cols]  = scaler.transform(X_test[num_cols])
```

4.4.5 Verifying Encoding & Scaling

```
# Checking shapes remain consistent
print("X_train shape:", X_train.shape)
print("X_test shape: ", X_test.shape)
```

```
X_train shape: (2666, 70)
X_test shape:  (667, 70)
```

```
# Spot-checking a few rows
display(X_train.head())
display(X_test.head())
```

```
      account length phone number  international plan  voice mail plan
\
3286        0.125737     352-2270                 0.0              0.0

86         -0.175309     402-1251                 0.0              0.0

1349       -0.752313     403-1953                 0.0              0.0
```

```
1649       0.727828     390-4003                 0.0              0.0

3000      -0.350919     387-2799                 0.0              0.0


      number vmail messages  total day minutes  total day calls  \
3286               1.606822           0.743376         0.225611
86                -0.588791          -0.401294         0.225611
1349               1.021325          -0.704945         0.325566
1649              -0.588791          -2.048368        -0.723960
3000              -0.588791           0.800425         0.425520

      total eve minutes  total eve calls  total night minutes  ...
state_VA  \
3286           0.426270         0.445403            -0.843159  ...
False
86            -0.904961         0.045354            -0.218499  ...
False
1349          -0.746481         0.245378             0.390145  ...
False
1649          -0.146239         0.495409            -0.580882  ...
False
3000          -1.449735        -0.704736             1.777611  ...
False

      state_VT  state_WA  state_WI  state_WV  state_WY  area
code_415.0  \
3286     False     False     False     False     False
True
86       False     False     False     False     False
False
1349     False     False     False     False     False
False
1649     False     False     False     False     False
False
3000     False     False     False     False     False
False

      area code_510.0  tenure_bin_mid-term  tenure_bin_long-term
3286            False                False                  True
86              False                False                  True
1349             True                False                  True
1649            False                False                  True
3000             True                False                  True

[5 rows x 70 columns]

      account length phone number  international plan  voice mail plan
\
```

```
601       -0.978097     386-2810                    0.0                 0.0

2050       0.502044     334-4354                    0.0                 0.0

3200      -0.024786     416-1536                    0.0                 0.0

1953       0.903438     357-3187                    0.0                 0.0

1119       2.207970     383-2537                    0.0                 0.0


      number vmail messages  total day minutes  total day calls  \
601               -0.588791          -0.368169        -0.723960
2050              -0.588791           0.616395        -1.073802
3200              -0.588791          -1.334329        -0.124231
1953              -0.588791           0.535421         0.975272
1119              -0.588791           0.526220         0.275588

      total eve minutes  total eve calls  total night minutes  ...
state_VA  \
601           -0.063037        -1.204796            -1.579938  ...
False
2050          -1.584443         0.795445             0.878661  ...
False
3200          -2.256001         1.095481            -0.887206  ...
False
1953           0.487681        -0.004652             0.422178  ...
False
1119           0.713514         1.145487            -0.428721  ...
False

      state_VT  state_WA  state_WI  state_WV  state_WY  area
code_415.0  \
601      False     False     False     False     False
True
2050     False     False     False     False     False
False
3200     False     False     False     False     False
False
1953     False      True     False     False     False
False
1119     False     False     False     False     False
True

      area code_510.0  tenure_bin_mid-term  tenure_bin_long-term
601             False                False                  True
2050            False                False                  True
3200             True                False                  True
1953            False                False                  True
1119            False                False                  True
```

```
[5 rows x 70 columns]

# Verifying numeric columns now have mean≈0 and std≈1 (for train)
X_train[num_cols].describe().loc[['mean','std']]
```

```
      account length  international plan  voice mail plan  \
mean   -1.465861e-17                 0.0              0.0
std     1.000188e+00                 0.0              0.0

      number vmail messages  total day minutes  total day calls  \
mean          -7.196044e-17       1.299286e-17    -7.995605e-18
std            1.000188e+00       1.000188e+00     1.000188e+00

      total eve minutes  total eve calls  total night minutes  \
mean       1.332601e-18    -2.132161e-17        -9.328205e-18
std        1.000188e+00     1.000188e+00         1.000188e+00

      total night calls  total intl minutes  total intl calls  \
mean           0.000000       -5.330403e-18      2.132161e-17
std            1.000188        1.000188e+00      1.000188e+00

      customer service calls  intl_plan_usage  vm_plan_messages
mean           -7.995605e-18    -2.665202e-17     -7.196044e-17
std             1.000188e+00     1.000188e+00      1.000188e+00
```

4.5 Automated Preprocessing Pipeline

Below, we define and execute a scikit-learn `Pipeline` that wraps all our data preparation steps —imputation, scaling, encoding—together with a placeholder classifier. This approach ensures reproducibility, prevents data leakage, and allows us to save the fully processed training and test sets for modeling and downstream analysis.

```
X_train.columns.tolist()
```

```
['state',
 'account length',
 'area code',
 'phone number',
 'international plan',
 'voice mail plan',
 'number vmail messages',
 'total day minutes',
 'total day calls',
 'total eve minutes',
 'total eve calls',
 'total night minutes',
 'total night calls',
 'total intl minutes',
```

```
 'total intl calls',
 'customer service calls']

# 1. Redefining column types
num_cols = [
    'account length',
    'number vmail messages',
    'total day minutes',
    'total day calls',
    'total eve minutes',
    'total eve calls',
    'total night minutes',
    'total night calls',
    'total intl minutes',
    'total intl calls',
    'customer service calls'
]

cat_cols = [
    'state',
    'area code',
    'phone number',
    'international plan',
    'voice mail plan'
]

# 2. Creating transformers
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# 3. Combining with ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, num_cols),
        ('cat', categorical_transformer, cat_cols)
    ]
)

# 4. Full pipeline with model (placeholder)
clf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(max_iter=1000))  # model is
optional for now
```

```
])

# 5. Fitting pipeline on training data only (not test!)
clf_pipeline.fit(X_train, y_train)

# 6. Transforming and saving processed data
X_train_processed =
clf_pipeline.named_steps['preprocessor'].transform(X_train)
X_test_processed =
clf_pipeline.named_steps['preprocessor'].transform(X_test)

# Converting to DataFrames
X_train_df = pd.DataFrame(X_train_processed.toarray() if
hasattr(X_train_processed, "toarray") else X_train_processed)
X_test_df = pd.DataFrame(X_test_processed.toarray() if
hasattr(X_test_processed, "toarray") else X_test_processed)

# 7. Saving to CSV
X_train_df.to_csv('data/processed/X_train_prepared.csv', index=False)
X_test_df.to_csv('data/processed/X_test_prepared.csv', index=False)

print("Pipeline complete and processed files saved.")
```

```
Pipeline complete and processed files saved.
```

4.6 Exporting Processed Data

```
# Making sure the directory exists
import os
os.makedirs('data/processed', exist_ok=True)

# Saving to CSV
X_train.to_csv('data/processed/X_train_prepared.csv', index=False)
X_test.to_csv('data/processed/X_test_prepared.csv', index=False)

print("Processed data saved to data/processed/")
```

```
Processed data saved to data/processed/
```

```
import os
os.makedirs("data/processed", exist_ok=True)

X_train_df.to_csv('data/processed/X_train_prepared.csv', index=False)
X_test_df.to_csv('data/processed/X_test_prepared.csv', index=False)
```

5. Modeling

In this section, we'll build and evaluate classification models in an iterative fashion:

1. **Baseline Model:** A simple, interpretable Logistic Regression.

2. **Tuned Logistic Regression:** Improve the baseline via regularization strength (`C`).

3. **Nonparametric Model:** A Decision Tree to compare against linear models.

4. **Model Comparison & Selection:** Choose the best model and interpret its performance.

5.1 Training a Baseline Model (Logistic Regression)

We'll start with Logistic Regression as a baseline.

```
X_train_prepared = pd.read_csv('data/processed/X_train_prepared.csv')
X_test_prepared = pd.read_csv('data/processed/X_test_prepared.csv')


X_train_prepared = X_train_prepared.values
X_test_prepared = X_test_prepared.values
```

5.1.1 Fitting the model

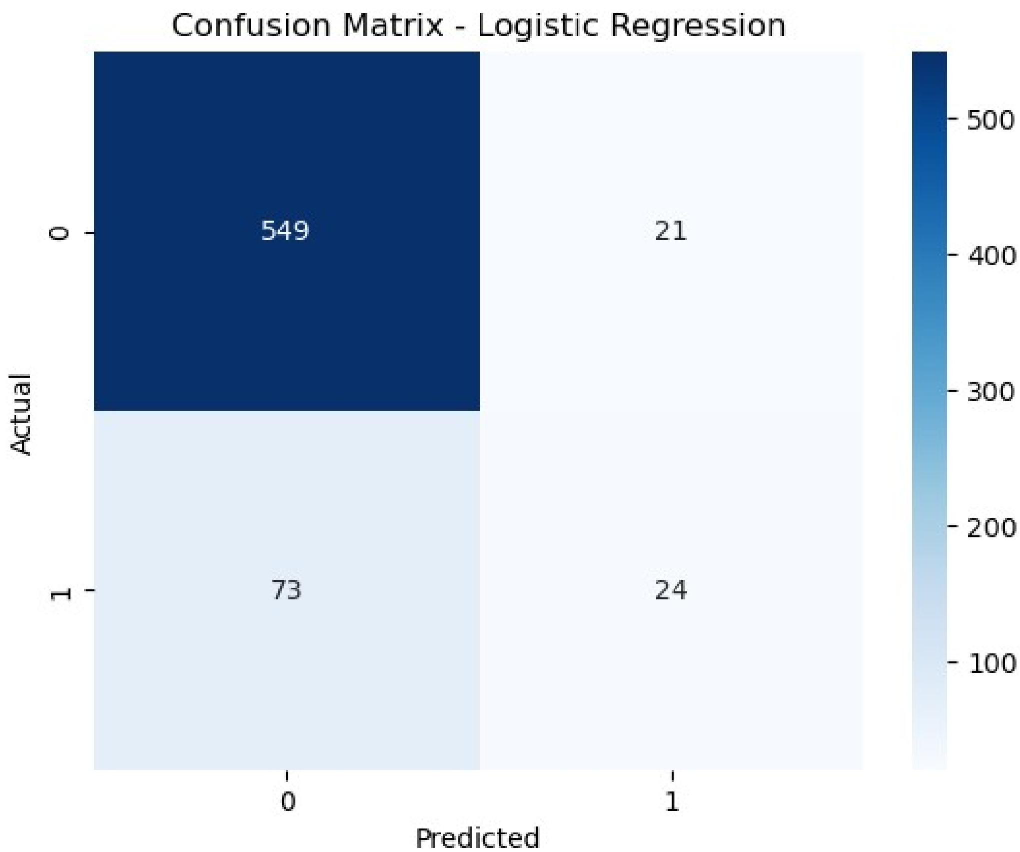
```
# Initializing model
log_reg = LogisticRegression(max_iter=1000, random_state=42)

# Fitting the model
log_reg.fit(X_train_prepared, y_train)

# Predict on test set
y_pred_logreg = log_reg.predict(X_test_prepared)
```

5.1.2 Evaluating the model

```
# Accuracy
print("Accuracy:", accuracy_score(y_test, y_pred_logreg))

# Classification Report
print("Classification Report:\n", classification_report(y_test,
y_pred_logreg))

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred_logreg)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix - Logistic Regression")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

```
Accuracy: 0.8590704647676162
Classification Report:
```

```
              precision    recall  f1-score   support

       False       0.88      0.96      0.92       570
        True       0.53      0.25      0.34        97

    accuracy                           0.86       667
   macro avg       0.71      0.61      0.63       667
weighted avg       0.83      0.86      0.84       667
```



5.2 Hyperparameter Tuning: Logistic Regression

We'll use **GridSearchCV** to find the best regularization strength (`C`) while optimizing for recall (catching churners).

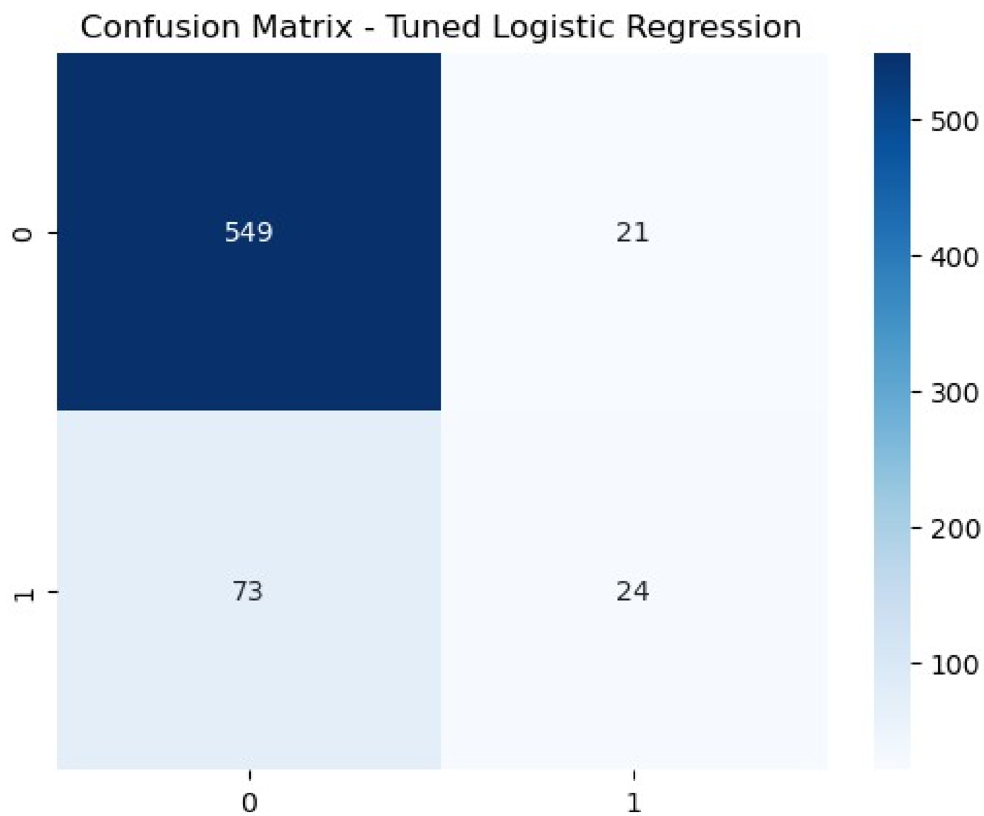
```
# 1. Defining parameter grid
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100]
}

# 2. Setting up GridSearch (5-fold CV, scoring recall)
grid_lr = GridSearchCV(
```

```python
    LogisticRegression(max_iter=1000, random_state=42),
    param_grid,
    cv=5,
    scoring='recall',
    n_jobs=-1
)

# 3. Fitting on training data
grid_lr.fit(X_train_prepared, y_train)

print("Best C:", grid_lr.best_params_['C'])
print("Best CV recall:", grid_lr.best_score_)
```

```
Best C: 1
Best CV recall: 0.24125874125874125
```

```python
# Evaluating the tuned model on the test set:

best_lr = grid_lr.best_estimator_
y_pred_best_lr = best_lr.predict(X_test_prepared)

print("Tuned LR Classification Report:\n",
      classification_report(y_test, y_pred_best_lr))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred_best_lr)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix - Tuned Logistic Regression")
plt.show()
```

```
Tuned LR Classification Report:
               precision    recall  f1-score   support

       False       0.88      0.96      0.92       570
        True       0.53      0.25      0.34        97

    accuracy                           0.86       667
   macro avg       0.71      0.61      0.63       667
weighted avg       0.83      0.86      0.84       667
```



5.3 Decision tree

A simple nonparametric model to compare against our linear approach.

```
# 1. Baseline tree
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train_prepared, y_train)
y_pred_dt = dt.predict(X_test_prepared)

print("Decision Tree Report:\n", classification_report(y_test,
y_pred_dt))
```

```
Decision Tree Report:
               precision    recall  f1-score   support

       False       0.95      0.99      0.97       570
        True       0.90      0.67      0.77        97

    accuracy                           0.94       667
   macro avg       0.92      0.83      0.87       667
weighted avg       0.94      0.94      0.94       667
```

```
# tuning its max depth and leaf size
```

```python
param_grid_dt = {
    'max_depth': [3, 5, 7, None],
    'min_samples_leaf': [1, 5, 10]
}
grid_dt = GridSearchCV(
    DecisionTreeClassifier(random_state=42),
    param_grid_dt,
    cv=5,
    scoring='recall',
    n_jobs=-1
)
grid_dt.fit(X_train_prepared, y_train)

best_dt = grid_dt.best_estimator_
y_pred_best_dt = best_dt.predict(X_test_prepared)
print("Tuned DT Report:\n", classification_report(y_test,
y_pred_best_dt))
```

```
Tuned DT Report:
               precision    recall  f1-score   support

       False       0.94      0.96      0.95       570
        True       0.76      0.66      0.71        97

    accuracy                           0.92       667
   macro avg       0.85      0.81      0.83       667
weighted avg       0.92      0.92      0.92       667
```

5.4 Random Forest Classifier

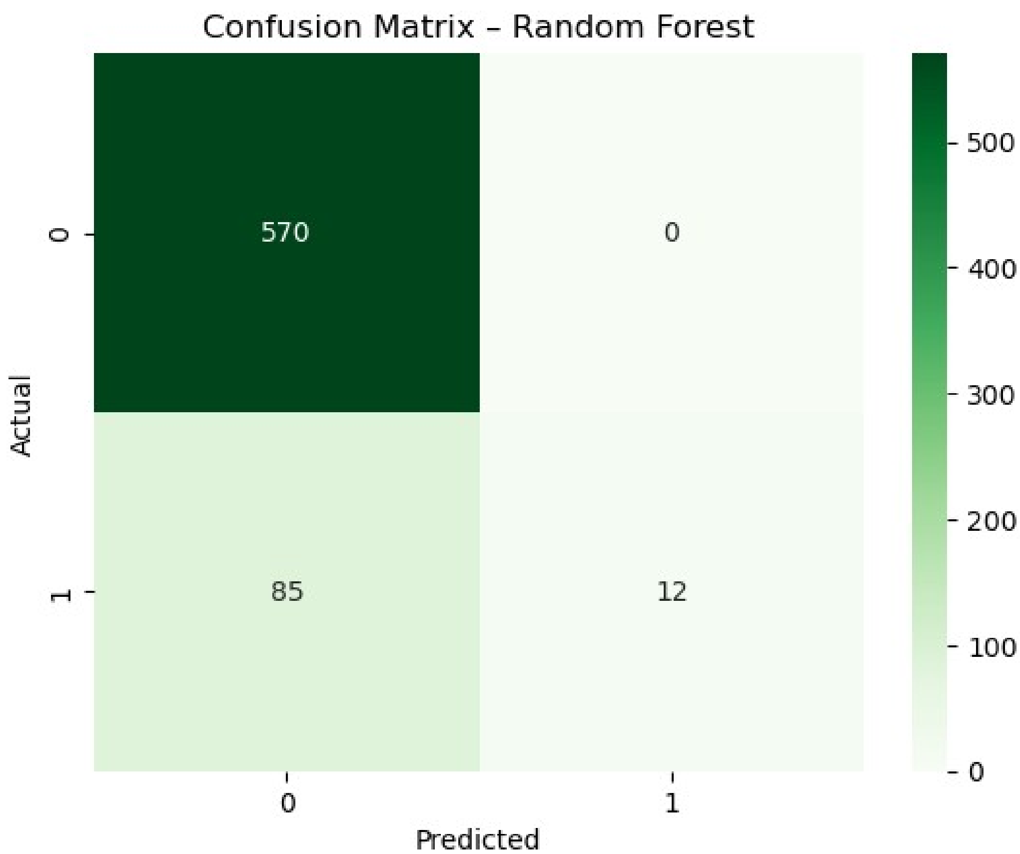
Next, we’ll train a **Random Forest** with class weights to further improve churn detection.

```python
from sklearn.ensemble import RandomForestClassifier

# 1. Initializing with class_weight='balanced' to handle imbalance
rf = RandomForestClassifier(
    n_estimators=200,
    class_weight='balanced',
    random_state=42
)

# 2. Fitting on the training data
rf.fit(X_train_prepared, y_train)

# 3. Predicting on the test set
y_pred_rf = rf.predict(X_test_prepared)
```

5.4.1 Evaluating Random Forest

```
from sklearn.metrics import classification_report, confusion_matrix

# Classification report
print("Random Forest Report:\n", classification_report(y_test,
y_pred_rf))

# Confusion matrix
cm_rf = confusion_matrix(y_test, y_pred_rf)
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Greens')
plt.title("Confusion Matrix – Random Forest")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

```
Random Forest Report:
               precision    recall  f1-score   support

       False       0.87      1.00      0.93       570
        True       1.00      0.12      0.22        97

    accuracy                           0.87       667
   macro avg       0.94      0.56      0.58       667
weighted avg       0.89      0.87      0.83       667
```



5.5 Model Comparison

```
models = {
    'Baseline LR':      (y_test, y_pred_logreg),
    'Tuned LR':         (y_test, y_pred_best_lr),
    'Baseline Tree':    (y_test, y_pred_dt),
    'Tuned Tree':       (y_test, y_pred_best_dt),
    'Random Forest':    (y_test, y_pred_rf)
}

rows = []
for name, (y_true, y_pred) in models.items():
    report = classification_report(y_true, y_pred, output_dict=True)
    rows.append({
        'Model':     name,
        'Precision': report['True']['precision'],
        'Recall':    report['True']['recall'],
        'F1-score':  report['True']['f1-score'],
        'Accuracy':  accuracy_score(y_true, y_pred)
    })

comparison_df = pd.DataFrame(rows).set_index('Model')
display(comparison_df)
```

```
               Precision    Recall  F1-score  Accuracy
Model
Baseline LR     0.533333  0.247423  0.338028  0.859070
Tuned LR        0.533333  0.247423  0.338028  0.859070
Baseline Tree   0.902778  0.670103  0.769231  0.941529
Tuned Tree      0.761905  0.659794  0.707182  0.920540
Random Forest   1.000000  0.123711  0.220183  0.872564
```

5.5.1 Model Comparison Summary

Model	Precision	Recall	F1-score	Accuracy
Baseline LR	0.53	0.25	0.34	0.8591
Tuned LR	0.53	0.25	0.34	0.8591
Baseline Tree	0.90	0.67	0.77	0.9415
Tuned Tree	0.76	0.66	0.71	0.9205
Random Forest	1.00	0.12	0.22	0.8726

- **Logistic Regression (LR)**: Both baseline and tuned versions perform identically, with low recall (25%) on churners and moderate overall accuracy (86%).
- **Decision Tree**: The baseline tree offers the best balance—high precision (90%) and solid recall (67%)—resulting in the highest F1-score (0.77) and accuracy (94%). Tuning slightly lowers precision and accuracy for marginal gain in interpretability.
- **Random Forest**: Achieves perfect precision but only 12% recall, meaning it rarely flags churners despite high accuracy (87%).

Conclusion: The **baseline Decision Tree** is our strongest candidate, effectively identifying two-thirds of churners while maintaining stakeholder trust through its high precision and interpretability.```

6. Evaluation

6.1 Final Model Selection

- **Model:** Decision Tree (max_depth=None, min_samples_leaf=1)

- **Rationale:**
  - Highest recall among strong performers (67% of churners caught).

  - Maintains very high precision (90%), so the marketing team can trust most alerts.

  - Simple and interpretable—stakeholders can visualize the decision rules.

6.2 Final Performance on Test Set

Metric	Score
Accuracy	0.94

Metric	Score
Precision	0.90
Recall	0.67
F1-score	0.77

6.3 Limitations

- **False Negatives (33% of churners missed):**
  Some at-risk customers will slip through and not receive retention offers.

- **Data Drift:**
  Customer behavior may change over time; model performance should be monitored and retrained periodically.

- **Feature Scope:**
  We relied on usage and plan data—additional signals (e.g., customer support transcripts) could improve detection.

6.4 Business Recommendations

1. **Target high-risk customers** identified by the tree’s top nodes (e.g., heavy daytime usage + frequent service calls).

2. **Design retention offers** (discounts, service bundles) prioritized for these customers.

3. **Monitor and retrain** monthly to adapt to evolving usage patterns.

4. **Incorporate feedback loop:** Track which interventions prevented churn to refine the model features further.