Hospital Length of Stay (LOS) Prediction

Context:

Hospital management is a vital area that gained a lot of attention during the COVID-19 pandemic. **Inefficient distribution of resources like beds, ventilators might lead to a lot of complications**. However, this can be mitigated by **predicting the length of stay (LOS) of a patient before getting admitted**. Once this is determined, the hospital can plan a suitable treatment, resources, and staff to reduce the LOS and increase the chances of recovery. The rooms and bed can also be planned in accordance with that.

HealthPlus hospital has been incurring a lot of losses in revenue and life due to its inefficient management system. They have been unsuccessful in allocating pieces of equipment, beds, and hospital staff fairly. **A system that could estimate the length of stay (LOS) of a patient can solve this problem to a great extent.**

Objective:

As a Data Scientist, you have been hired by HealthPlus to analyze the data, find out **what factors affect the LOS the most, and come up with a machine learning model which can predict the LOS of a patient** using the data available during admission and after running a few tests. Also, **bring about useful insights and policies from the data, which can help the hospital to improve their health care infrastructure and revenue.**

Data Dictionary:

The data contains various information recorded during the time of admission of the patient. It only contains **records of patients who were admitted to the hospital.** The detailed data dictionary is given below:

- **patientid**: Patient ID
- **Age**: Range of age of the patient
- **gender**: Gender of the patient
- **Type of Admission**: Trauma, emergency or urgent
- **Severity of Illness**: Extreme, moderate, or minor
- **health_condition**s: Any previous health conditions suffered by the patient
- **Visitors with Patient**: The number of patients who accompany the patient
- **Insurance**: Does the patient have health insurance or not?
- **Admission_Deposit**: The deposit paid by the patient during admission

- **Stay (in days)**: The number of days that the patient has stayed in the hospital. This is the **target variable**
- **Available Extra Rooms in Hospital**: The number of rooms available during admission
- **Department**: The department which will be treating the patient
- **Ward_Facility_Code**: The code of the ward facility in which the patient will be admitted
- **doctor_name**: The doctor who will be treating the patient
- **staff_available**: The number of staff who are not occupied at the moment in the ward

Approach to solve the problem:

1. Import the necessary libraries
2. Read the dataset and get an overview
3. Exploratory data analysis - a. Univariate b. Bivariate
4. Data preprocessing if any
5. Define the performance metric and build ML models
6. Checking for assumptions
7. Compare models and determine the best one
8. Observations and business insights

Importing Libraries

In [7]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings("ignore")

# Removes the limit for the number of displayed columns
pd.set_option("display.max_columns", None)

# Sets the limit for the number of displayed rows
pd.set_option("display.max_rows", 200)

# To build models for prediction
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor,BaggingRegressor

# To encode categorical variables
from sklearn.preprocessing import LabelEncoder
```

```
# For tuning the model
from sklearn.model_selection import GridSearchCV

# To check model performance
from sklearn.metrics import make_scorer,mean_squared_error, r2_score, mean_a
```

In [8]:
```
# Read the healthcare dataset file
data = pd.read_csv("healthcare_data.csv")
```

In [9]:
```
# Copying data to another variable to avoid any changes to original data
same_data = data.copy()
```

Data Overview

In [10]:
```
# View the first 5 rows of the dataset
data.head()
```

Out[10]:

	Available Extra Rooms in Hospital	Department	Ward_Facility_Code	doctor_name	staff_available	patientid
0	4	gynecology	D	Dr Sophia	0	33070
1	4	gynecology	B	Dr Sophia	2	34808
2	2	gynecology	B	Dr Sophia	8	44577
3	4	gynecology	D	Dr Olivia	7	3695
4	2	anesthesia	E	Dr Mark	10	108956

In [11]:
```
# View the last 5 rows of the dataset
data.tail()
```

Out[11]:

	Available Extra Rooms in Hospital	Department	Ward_Facility_Code	doctor_name	staff_available	pa
499995	4	gynecology	F	Dr Sarah	2	
499996	13	gynecology	F	Dr Olivia	8	
499997	2	gynecology	B	Dr Sarah	3	
499998	2	radiotherapy	A	Dr John	1	
499999	3	gynecology	F	Dr Sophia	3	

In [12]:
```
# Understand the shape of the data
data.shape
```

Out[12]:
```
(500000, 15)
```

- The dataset has **5,00,000 rows and 15 columns.**

In [13]:
```
# Checking the info of the data
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500000 entries, 0 to 499999
Data columns (total 15 columns):
 #   Column                            Non-Null Count   Dtype  
---  ------                            --------------   -----  
 0   Available Extra Rooms in Hospital  500000 non-null  int64  
 1   Department                        500000 non-null  object 
 2   Ward_Facility_Code                500000 non-null  object 
 3   doctor_name                       500000 non-null  object 
 4   staff_available                   500000 non-null  int64  
 5   patientid                         500000 non-null  int64  
 6   Age                               500000 non-null  object 
 7   gender                            500000 non-null  object 
 8   Type of Admission                 500000 non-null  object 
 9   Severity of Illness               500000 non-null  object 
 10  health_conditions                 348112 non-null  object 
 11  Visitors with Patient             500000 non-null  int64  
 12  Insurance                         500000 non-null  object 
 13  Admission_Deposit                 500000 non-null  float64
 14  Stay (in days)                    500000 non-null  int64  
dtypes: float64(1), int64(5), object(9)
memory usage: 57.2+ MB
```

Observations:

- Available Extra Rooms in Hospital, staff_available, patientid, Visitors with Patient, Admission_Deposit, and Stay (in days) are of **numeric data type** and the rest of the columns are of **object data type**.
- The number of non-null values is the same as the total number of entries in the data, i.e., **there are no null values.**
- The column patientid is an identifier for patients in the data. This column will not help with our analysis so we can drop it.

In [14]:
```
# To view patientid and the number of times they have been admitted to the h
data['patientid'].value_counts()
```

Out[14]:
```
patientid
126719    21
125695    21
44572     21
126623    21
125625    19
          ..
37634      1
91436      1
118936     1
52366      1
105506     1
Name: count, Length: 126399, dtype: int64
```

Observation:

- **The maximum number of times the same patient admitted to the hospital is 21 and minimum is 1.**

In [15]:
```
# Dropping patientid from the data as it is an identifier and will not add v
data=data.drop(columns=["patientid"])
```

In [16]:
```
# Checking for duplicate values in the data
data.duplicated().sum()
```

Out[16]:
```
0
```

Observation:

- Data contains unique rows. There is no need to remove any rows.

In [17]:
```
# Checking the descriptive statistics of the columns
data.describe().T
```

Out[17]:

	count	mean	std	min	25%	
Available Extra Rooms in Hospital	500000.0	3.638800	2.698124	0.000000	2.000000	
staff_available	500000.0	5.020470	3.158103	0.000000	2.000000	
Visitors with Patient	500000.0	3.549414	2.241054	0.000000	2.000000	
Admission_Deposit	500000.0	4722.315734	1047.324220	1654.005148	4071.714532	46
Stay (in days)	500000.0	12.381062	7.913174	3.000000	8.000000	

Observations:

- There are around **3 rooms available in the hospital on average** and there are times when the hospital is full and there are no rooms available (minimum value is 0). The **maximum number of rooms available in the hospital is 24**.
- **On average, there are around 5 staff personnel available to treat the new patients** but it can also be zero at times. The maximum number of staff available in the hospital is 10.
- **On average, around 3 visitors accompany the patient.** Some patients come on their own (minimum value is zero) and a few cases have 32 visitors. It will be interesting to see if there is any relationship between the number of visitors and the severity of the patient.
- **The average admission deposit lies around 4,722 dollars and a minimum of 1,654 dollars is paid on every admission.**
- **Patient's stay ranges from 3 to 51 days.** There might be outliers in this variable. The median length of stay is 9 days.

In [18]:

```
# List of all important categorical variables
cat_col = ["Department", "Type of Admission", 'Severity of Illness', 'gender

# Printing the number of occurrences of each unique value in each categorica
for column in cat_col:
    print(data[column].value_counts(1))
    print("-" * 50)
```

```
Department
gynecology            0.686956
radiotherapy          0.168630
anesthesia            0.088358
TB & Chest disease    0.045780
surgery               0.010276
Name: proportion, dtype: float64
--------------------------------------------------
Type of Admission
Trauma       0.621072
Emergency    0.271568
Urgent       0.107360
Name: proportion, dtype: float64
--------------------------------------------------
Severity of Illness
Moderate    0.560394
Minor       0.263074
Extreme     0.176532
Name: proportion, dtype: float64
--------------------------------------------------
gender
Female    0.74162
Male      0.20696
Other     0.05142
Name: proportion, dtype: float64
--------------------------------------------------
Insurance
Yes    0.78592
No     0.21408
Name: proportion, dtype: float64
--------------------------------------------------
health_conditions
Other                   0.271209
High Blood Pressure    0.228093
Diabetes               0.211553
Asthama                0.188198
Heart disease          0.100947
Name: proportion, dtype: float64
--------------------------------------------------
doctor_name
Dr Sarah     0.199192
Dr Olivia    0.196704
Dr Sophia    0.149506
Dr Nathan    0.141554
Dr Sam       0.111422
Dr John      0.102526
Dr Mark      0.088820
Dr Isaac     0.006718
Dr Simon     0.003558
Name: proportion, dtype: float64
--------------------------------------------------
Ward_Facility_Code
F    0.241076
D    0.238110
B    0.207770
E    0.190748
```

```
A    0.093102
C    0.029194
Name: proportion, dtype: float64
------------------------------------------------
Age
21-30     0.319586
31-40     0.266746
41-50     0.160812
11-20     0.093072
61-70     0.053112
51-60     0.043436
71-80     0.037406
81-90     0.016362
0-10      0.006736
91-100    0.002732
Name: proportion, dtype: float64
------------------------------------------------
```

Observations:

- **The majority of patients (~82%) admit to the hospital with moderate and minor illness**, which is understandable as extreme illness is less frequent than moderate and minor illness.
- **Gynecology department gets the most number of patients (~68%)** in the hospital, whereas patients in Surgery department are very few (~1%).
- **Ward A and C accommodate the least number of patients (~12%).** These might be wards reserved for patient with extreme illness and patients who need surgery. It would be interesting to see if patients from these wards also stay for longer duration.
- **The majority of patients belong to the age group of 21-50 (~75%), and the majority of patients are women (~74%).** The most number of patients in the gynecology department of the hospital can justify this.
- Most of the patients admitted to the hospital are the cases of trauma (~62%).
- After 'Other' category, **High Blood Pressure and Diabetes are the most common health conditions.**

Exploratory Data Analysis (EDA)

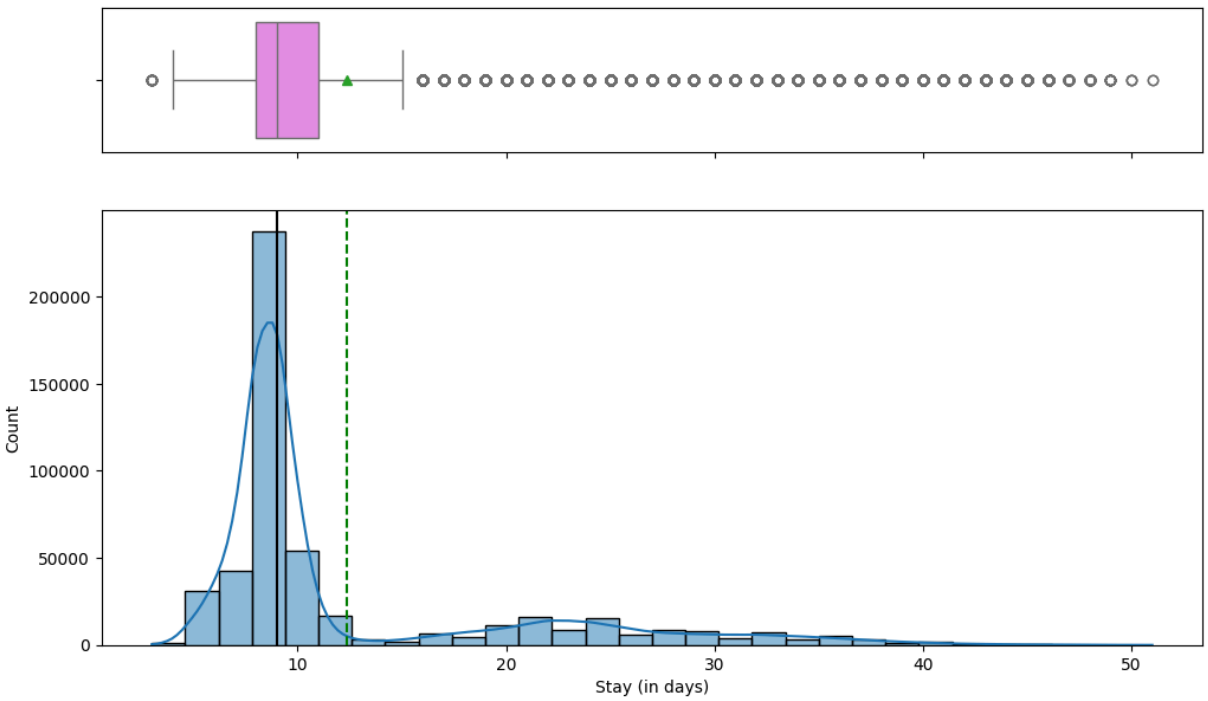
Univariate Analysis

In [19]:
```
# Function to plot a boxplot and a histogram along the same scale

def histogram_boxplot(data, feature, figsize=(12, 7), kde=False, bins=None):
    """
    Boxplot and histogram combined

    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
```

```
    kde: whether to the show density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows = 2,       # Number of rows of the subplot grid = 2
        sharex = True,  # x-axis will be shared among all subplots
        gridspec_kw = {"height_ratios": (0.25, 0.75)},
        figsize = figsize,
    )                    # Creating the 2 subplots
    sns.boxplot(data = data, x = feature, ax = ax_box2, showmeans = True, cc
    )                    # Boxplot will be created and a star will indicate t
    sns.histplot(
        data = data, x = feature, kde = kde, ax = ax_hist2, bins = bins, pal
    ) if bins else sns.histplot(
        data = data, x = feature, kde = kde, ax = ax_hist2
    )                    # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color = "green", linestyle = "--"
    )                    # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color = "black", linestyle = "-"
    )                    # Add median to the histogram
```

Length of stay

In [20]:
```
histogram_boxplot(data, "Stay (in days)", kde = True, bins = 30)
```



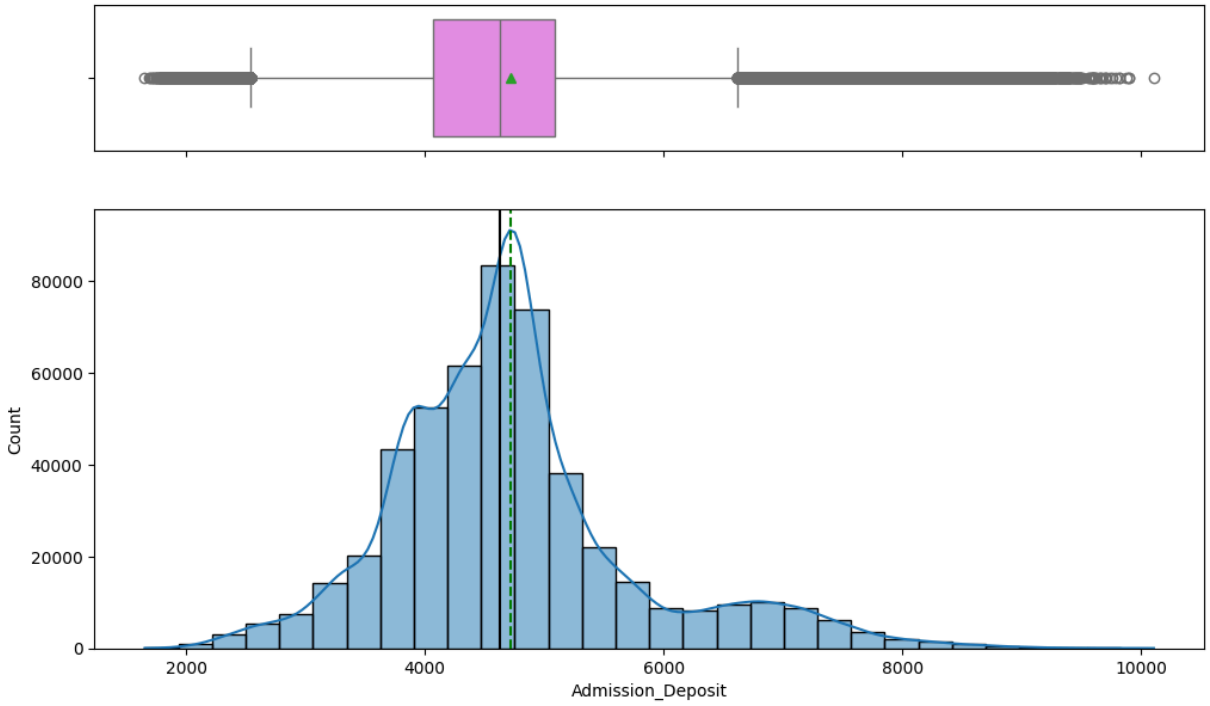
Observations:

- **Fewer patients are staying more than 10 days in the hospital and very few stay for more than 40 days**. This might be because the majority of patients are admitted for moderate or minor illnesses.

- The peak of the distribution shows that **most of the patients stay for 8-9 days in the hospital.**

Admission Deposit

In [21]:
```
histogram_boxplot(data, "Admission_Deposit", kde = True, bins = 30)
```

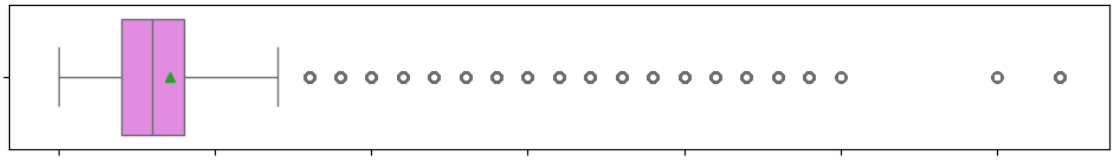


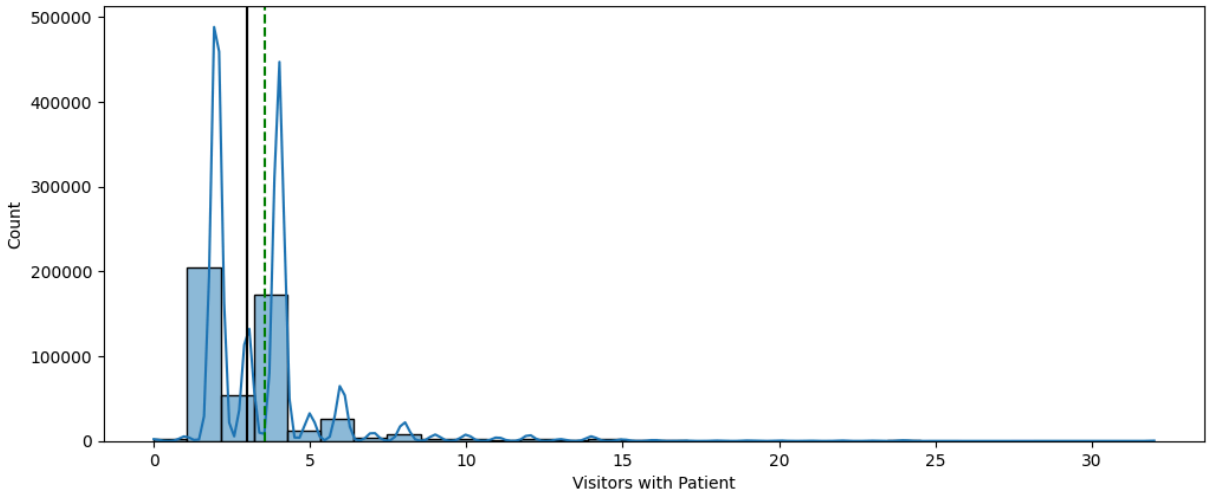
Observation:

- The **distribution of admission fees is close to normal with outliers on both sides**. Few patients are paying a high amount of admission fees and few patients are paying a low amount of admission fees.

Visitors with Patients

In [22]:
```
histogram_boxplot(data, "Visitors with Patient", kde = True, bins = 30)
```



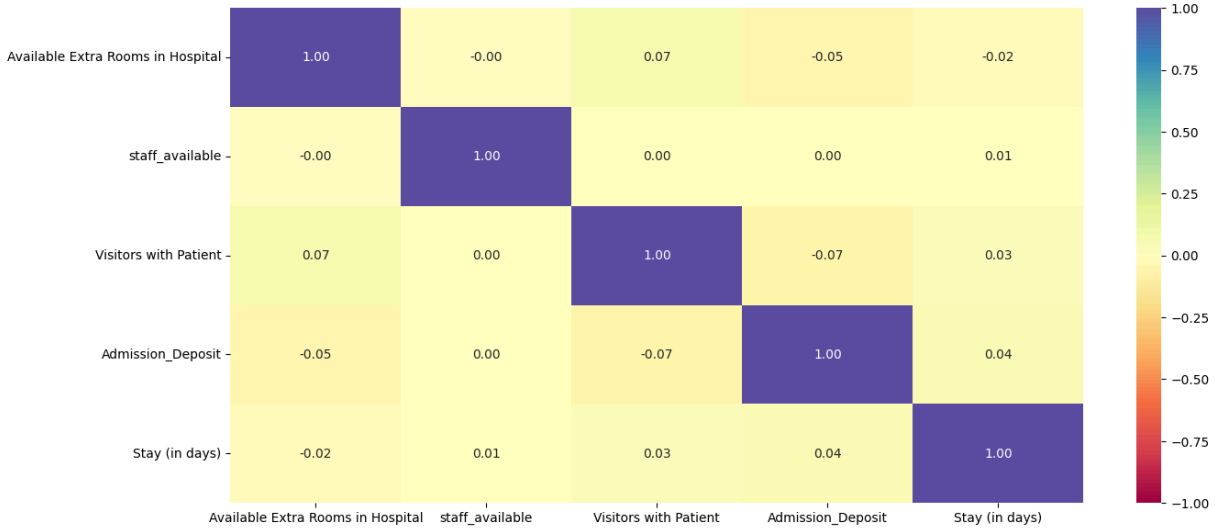


Observations:

- The distribution of the number of visitors with the patient is **highly skewed towards the right**.
- **2 and 4 are the most common number of visitors with patients.**

Bivariate Analysis

In [23]:
```
numeric_data = data.select_dtypes(include=[np.number])

# Finding the correlation between various columns of the dataset
plt.figure(figsize=(15, 7))
sns.heatmap(numeric_data.corr(), annot=True, vmin=-1, vmax=1, fmt=".2f", cma
plt.show()
```



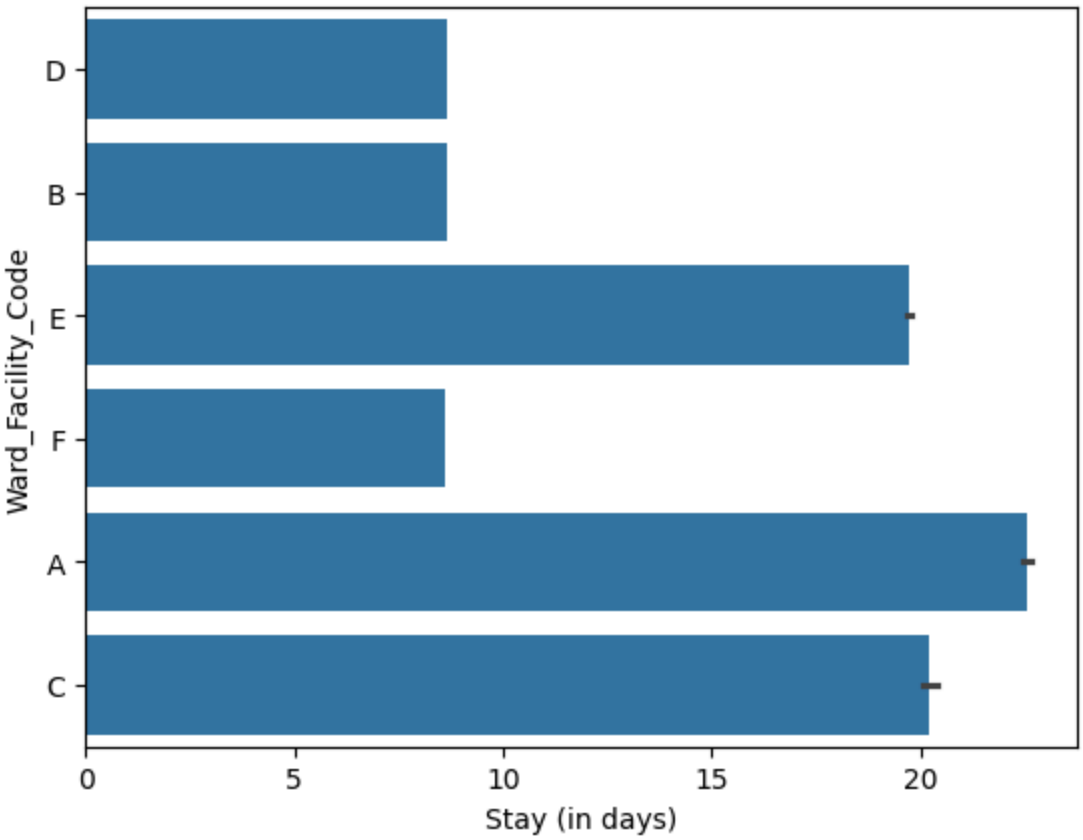
Observations:

- The heatmap shows that there is **no correlation between variables**.
- The continuous variables show no correlation with the target variable (Stay (in days)), which indicates that the **categorical variables might be more important for the prediction.**

In [24]:
```
# Function to plot stacked bar plots

def stacked_barplot(data, predictor, target):
    """
    Print the category counts and plot a stacked bar chart

    data: dataframe
    predictor: independent variable
    target: target variable
    """
    count = data[predictor].nunique()
    sorter = data[target].value_counts().index[-1]
    tab1 = pd.crosstab(data[predictor], data[target], margins = True).sort_v
        by = sorter, ascending = False
    )
    print(tab1)
    print("-" * 120)
    tab = pd.crosstab(data[predictor], data[target], normalize = "index").so
        by = sorter, ascending = False
    )
    tab.plot(kind = "bar", stacked = True, figsize = (count + 1, 5))
    plt.legend(
        loc = "lower left",
        frameon = False,
    )
    plt.legend(loc = "upper left", bbox_to_anchor = (1, 1))
    plt.show()
```

Let's start by checking the distribution of the LOS for the various wards

In [25]:
```
sns.barplot(y = 'Ward_Facility_Code', x = 'Stay (in days)', data = data)
plt.show()
```

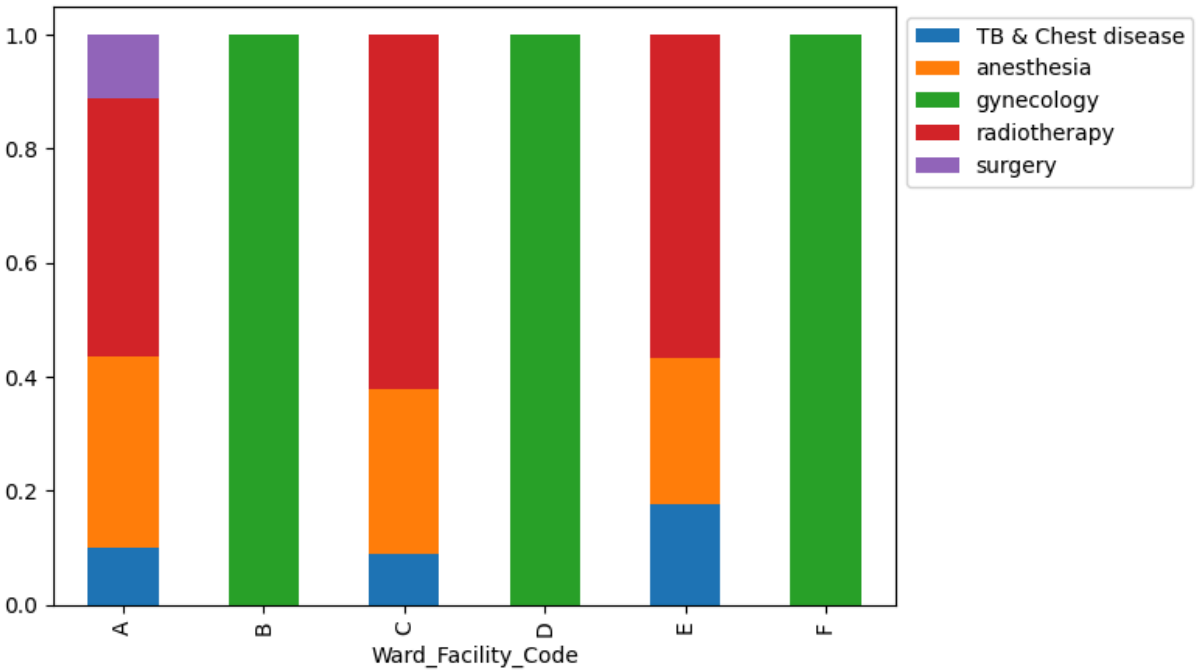


Observation:

- The hypothesis we made earlier is correct, i.e., **wards A and C has the patients staying for the longest duration, which implies these wards might be for patients with serious illnesses.**

In [26]:
```
stacked_barplot(data, "Ward_Facility_Code", "Department")
```

```
Department          TB & Chest disease  anesthesia  gynecology  radiotherapy
\
Ward_Facility_Code
A                                 4709       15611           0         21093
All                              22890       44179      343478         84315
B                                    0           0      103885             0
C                                 1319        4199           0          9079
D                                    0           0      119055             0
E                                16862       24369           0         54143
F                                    0           0      120538             0

Department          surgery     All
Ward_Facility_Code
A                      5138   46551
All                    5138  500000
B                         0  103885
C                         0   14597
D                         0  119055
E                         0   95374
F                         0  120538
```



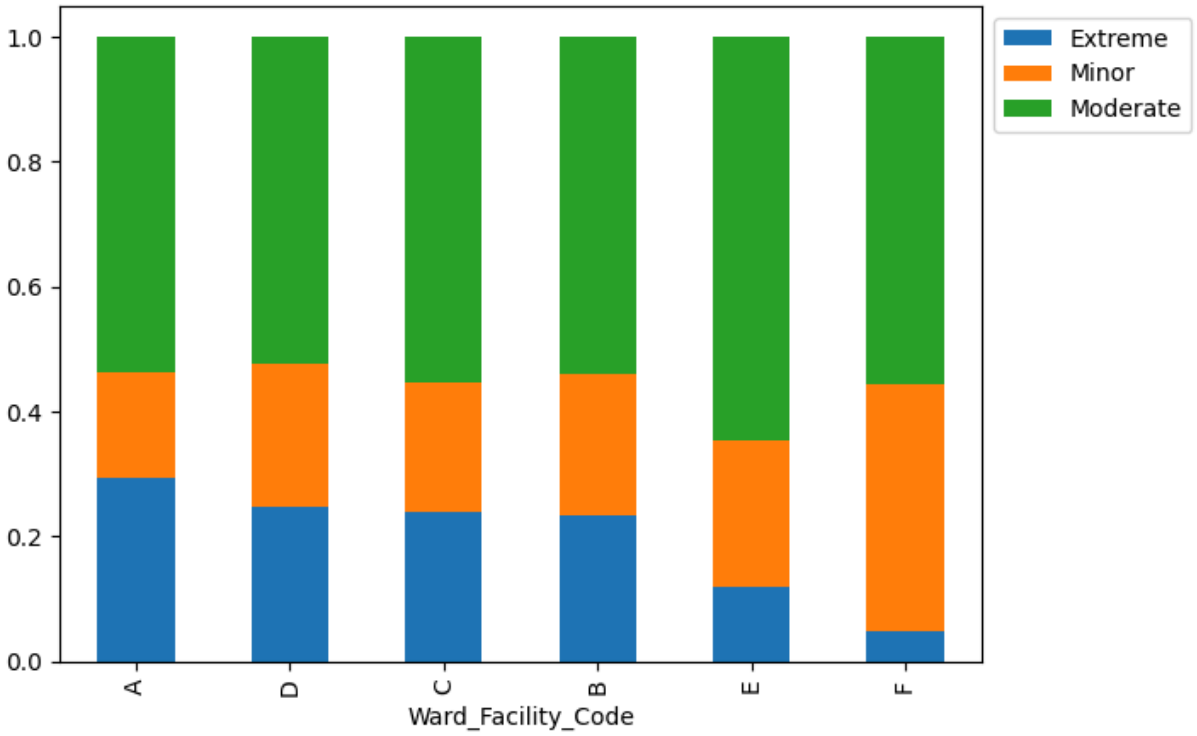
Observations:

- **Ward Facility B, D, and F are dedicated only to the gynecology department.**
- Wards A, C, and E have patients with all other diseases, and **patients undergoing surgery are admitted to ward A only.**

Usually, the more severe the illness, the more the LOS, let's check the distribution of severe patients in various wards.

In [27]:
```
stacked_barplot(data, "Ward_Facility_Code", "Severity of Illness")
```

```
Severity of Illness  Extreme   Minor  Moderate     All
Ward_Facility_Code
All                    88266  131537    280197  500000
D                      29549   27220     62286  119055
B                      24222   23579     56084  103885
A                      13662    7877     25012   46551
E                      11488   22254     61632   95374
F                       5842   47594     67102  120538
C                       3503    3013      8081   14597
```

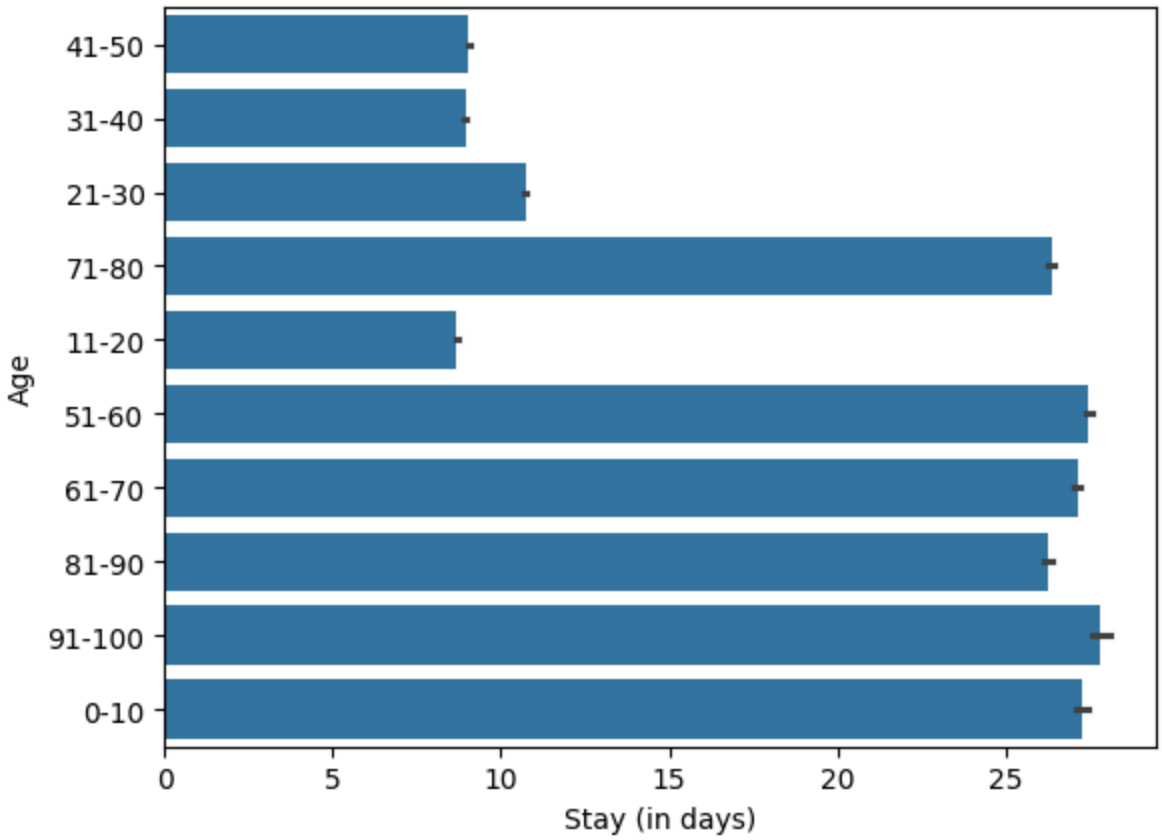


Observations:

- **Ward A has the highest number of extreme cases.** We observed earlier that ward A has the longest length of stay in the hospital as well. It might require more staff and resources as compared to other wards.
- **Ward F has the highest number of minor cases and Ward E has the highest number of moderate cases.**

Age can also be an important factor to find the length of stay. Let's check the same.

In [28]:
```
sns.barplot(y = 'Age', x = 'Stay (in days)', data = data)
plt.show()
```



Observation:

- **Patients aged between 1-10 and 51-100 tend to stay the most number of days in the hospital.** This might be because the majority of the patients between the 21-50 age group get admitted to the gynecology department and patients in age groups 1-10 and 5-100 might get admitted due to some serious illness.

Let's look at the doctors, their department names, and the total number of patients they have treated.

In [29]:
```
data.groupby(['doctor_name'])['Department'].agg(Department_Name='unique',Pat
```

Out[29]:

doctor_name	Department_Name	Patients_Treated
Dr Isaac	[surgery]	3359
Dr John	[TB & Chest disease, anesthesia, radiotherapy]	51263
Dr Mark	[anesthesia, TB & Chest disease]	44410
Dr Nathan	[gynecology]	70777
Dr Olivia	[gynecology]	98352
Dr Sam	[radiotherapy]	55711
Dr Sarah	[gynecology]	99596
Dr Simon	[surgery]	1779
Dr Sophia	[gynecology]	74753

Observations:

- The hospital employs a total of 9 doctors. Four of the doctors work in the department of gynecology, which sees the most patients.
- The majority of patients that attended the hospital were treated by Dr. Sarah and Olivia.
- Two doctors are working in the surgical department (Dr. Isaac and Dr. Simon), while Dr. Sam works in the radiotherapy department.
- The only two doctors who work in several departments are Dr. John and Dr. Mark.

Data Preparation for Model Building

- Before we proceed to build a model, we'll have to encode categorical features.
- Separate the independent variables and dependent Variables.
- We'll split the data into train and test to be able to evaluate the model that we train on the training data.

In [30]:
```
# Creating dummy variables for the categorical columns
# drop_first=True is used to avoid redundant variables
data = pd.get_dummies(
    data,
    columns = data.select_dtypes(include = ["object", "category"]).columns.t
    drop_first = True,
)
```

In [31]:
```
# Check the data after handling categorical data
data
```

Out[31]:

	Available Extra Rooms in Hospital	staff_available	Visitors with Patient	Admission_Deposit	Stay (in days)	Department_a
0	4	0	4	2966.408696	8	
1	4	2	2	3554.835677	9	
2	2	8	2	5624.733654	7	
3	4	7	4	4814.149231	8	
4	2	10	2	5169.269637	34	
...	...	...	...	...	...	
499995	4	2	3	4105.795901	10	
499996	13	8	2	4631.550257	11	
499997	2	3	2	5456.930075	8	
499998	2	1	2	4694.127772	23	
499999	3	3	4	4713.868519	10	

500000 rows × 42 columns

In [32]:
```
# Separating independent variables and the target variable
x = data.drop('Stay (in days)',axis=1)

y = data['Stay (in days)']
```

In [33]:
```
# Splitting the dataset into train and test datasets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, s
```

In [34]:
```
# Checking the shape of the train and test data
print("Shape of Training set : ", x_train.shape)
print("Shape of test set : ", x_test.shape)
```

```
Shape of Training set :  (400000, 41)
Shape of test set :  (100000, 41)
```

Model Building

- We will be using different metrics functions defined in sklearn like RMSE, MAE, $R2$, Adjusted $R2$, and MAPE for regression model evaluation. We will define a function to calculate these metrics.
- The mean absolute percentage error (MAPE) measures the accuracy of predictions as a percentage and can be calculated as the average absolute percentage error for all data points. The absolute percentage error is defined as the predicted value

minus actual value divided by actual value. It works best if there are no extreme values in the data and none of the actual values are 0.

In [35]:
```
# Function to compute adjusted R-squared
def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))


# Function to compute MAPE
def mape_score(targets, predictions):
    return np.mean(np.abs(targets - predictions) / targets) * 100


# Function to compute different metrics to check performance of a regressior
def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performa

    model: regressor
    predictors: independent variables
    target: dependent variable
    """

    pred = model.predict(predictors)                    # Predict using the ir
    r2 = r2_score(target, pred)                         # To compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)      # To compute adjusted
    rmse = np.sqrt(mean_squared_error(target, pred))    # To compute RMSE
    mae = mean_absolute_error(target, pred)             # To compute MAE
    mape = mape_score(target, pred)                     # To compute MAPE

    # Creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )

    return df_perf
```

Decision Trees

What is a Decision Tree?

Decision Trees are a type of supervised machine learning algorithm that can be used for **both classification and regression tasks**. They are often used in business and industry

to make decisions based on data, and are particularly useful for tasks that require decision-making based on a set of conditions.

How does a Decision Tree work?

A Decision Tree works by recursively splitting the dataset into smaller subsets based on the feature that provides the most information gain at each step. This process continues until the subsets are as pure as possible, meaning that they contain as few mixed class labels as possible, or until a stopping criterion is met (e.g., when a maximum depth is reached).

$$Information\ Gain = Entropy\ before\ split - Entropy\ after\ split$$

where,

$$Entropy = -\sum_{i=1}^{c} p_i \log_2 p_i$$

Here, p is the proportion of positive instances in the subset.

The goal of the algorithm is to find the tree that provides the best predictions on the training data, while also being as simple and interpretable as possible.

In [36]:
```
# Decision Tree Regressor
dt_regressor = DecisionTreeRegressor(random_state = 1)

# Fitting the model
dt_regressor.fit(x_train, y_train)

# Model Performance on the test data, i.e., prediction
dt_regressor_perf_test = model_performance_regression(dt_regressor, x_test,

dt_regressor_perf_test
```

Out[36]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	1.821321	1.13127	0.947324	0.947302	9.353216

Let's visualize the decision tree and examine the tree's decision rules. Visualizing a Decision Tree can help you understand how the algorithm works and interpret its predictions. Visualizing the tree can help us to:

- **Identify the root node:** The first node at the top of the tree is called the root node. It represents the entire dataset and is used to split the data into two or more homogeneous subsets.
- **Identify the internal nodes:** The nodes that are not leaf nodes are called internal nodes. They represent a decision or a test on a feature and are used to split the data

into smaller subsets based on the feature value.

- **Identify the leaf nodes:** The nodes at the bottom of the tree are called leaf nodes. They represent the output or the class label of the data after going through all the splits in the tree.

- **Follow a path from the root to a leaf node:** To interpret a decision tree, you can follow a path from the root node to a leaf node. Along the path, you can see the tests performed on the features, and based on the test results, the data is split into smaller subsets.

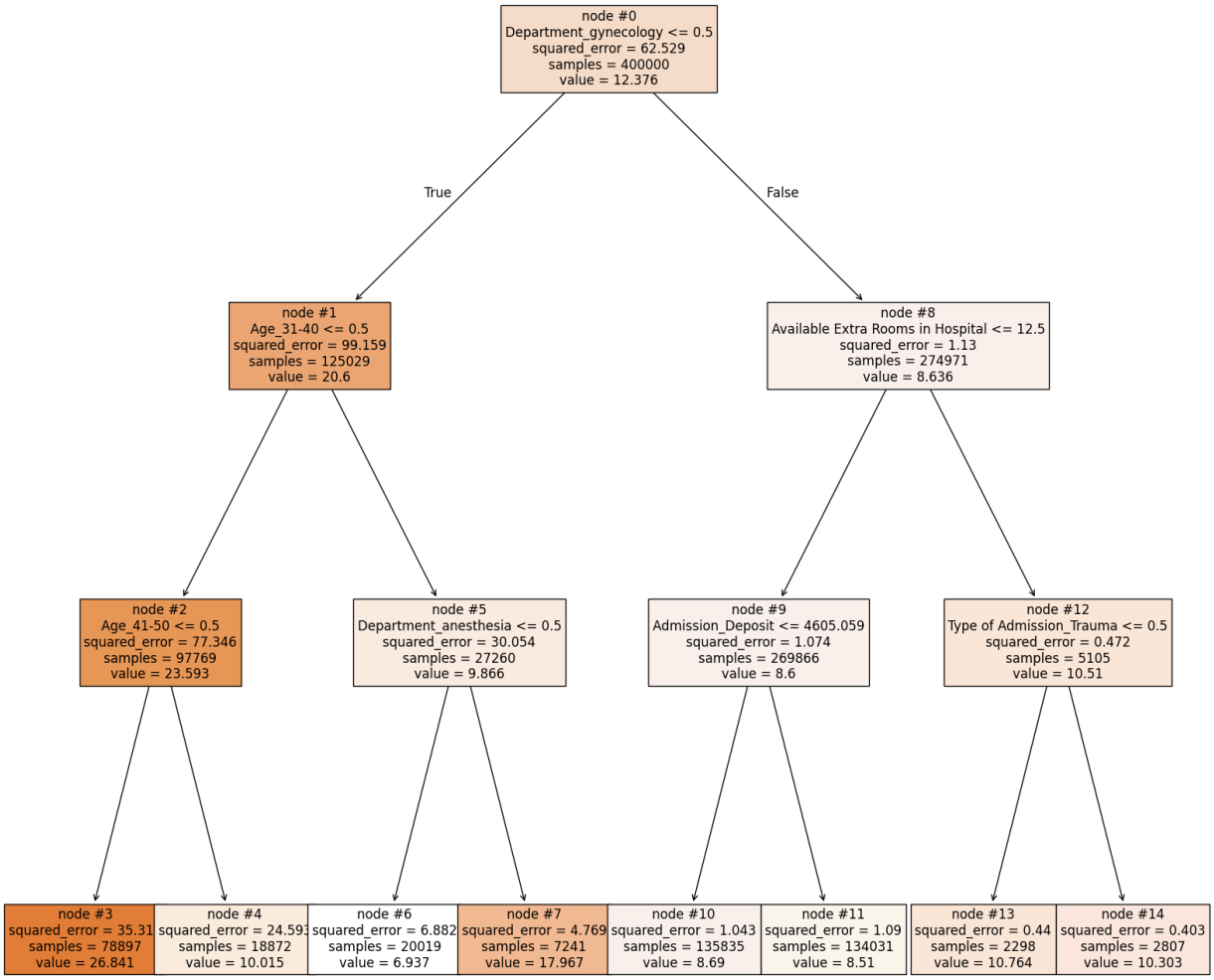
- **Analyze the feature importance:** You can analyze the feature importance by looking at the splits in the tree. The features used to split the data at the top of the tree are the most important features, as they have the highest impact on the decision.

- **Analyze the class distribution:** You can analyze the class distribution at the leaf nodes to understand how the decision tree predicts the class labels. If the majority of the samples in a leaf node belong to a particular class, the decision tree predicts that class for the new data.

- **Explain the decision:** Finally, you can explain the decision made by the decision tree by summarizing the path taken from the root to the leaf node and the class label predicted at the leaf node. You can also explain the importance of the features used in the decision and how they influence the final prediction.

We will limit the decision tree's depth to two so that we can visualize it better.

In [37]:
```
from sklearn import tree
features = list(x.columns)

# Building the model with max_depth=3
dt_regressor_visualize = DecisionTreeRegressor(random_state = 1, max_depth=3

# Fitting the model
dt_regressor_visualize.fit(x_train, y_train)


plt.figure(figsize = (20, 20))
tree.plot_tree(dt_regressor_visualize, feature_names = features, filled = Tr
               node_ids = True, class_names = True)
plt.show()
```



In [38]:
```
print(tree.export_text(dt_regressor_visualize, feature_names=x_train.columns
```

```
|--- Department_gynecology <= 0.50
|   |--- Age_31-40 <= 0.50
|   |   |--- Age_41-50 <= 0.50
|   |   |   |--- value: [26.84]
|   |   |--- Age_41-50 >  0.50
|   |   |   |--- value: [10.02]
|   |--- Age_31-40 >  0.50
|   |   |--- Department_anesthesia <= 0.50
|   |   |   |--- value: [6.94]
|   |   |--- Department_anesthesia >  0.50
|   |   |   |--- value: [17.97]
|--- Department_gynecology >  0.50
|   |--- Available Extra Rooms in Hospital <= 12.50
|   |   |--- Admission_Deposit <= 4605.06
|   |   |   |--- value: [8.69]
|   |   |--- Admission_Deposit >  4605.06
|   |   |   |--- value: [8.51]
|   |--- Available Extra Rooms in Hospital >  12.50
|   |   |--- Type of Admission_Trauma <= 0.50
|   |   |   |--- value: [10.76]
|   |   |--- Type of Admission_Trauma >  0.50
|   |   |   |--- value: [10.30]
```

Observations:

- **Root Node:** Department_gynecology <= 0.5. This is the starting point of the decision tree, which means that the Gynecology department results in the highest information gain among all the features. If the value is less than or equal to 0.5, the left branch is taken, and if it is greater than 0.5, the right branch is taken.
- **Internal Nodes:**
  - Age_31-40 <= 0.5
  - Age_41-50 <= 0.5
  - Department_anesthesia <= 0.5
  - Available Extra Rooms in Hospital <= 12.5
  - Admission_Deposit <= 4605.06
  - Type of Admission_Trauma <= 0.5

These are the intermediate nodes of the tree. Each node represents a decision based on a particular feature and a threshold value. Depending on the value of the feature, the tree follows the appropriate branch until it reaches a leaf node.

- **Leaf nodes** are the nodes in the tree that do not have any child nodes. In this tree, the leaf nodes correspond to the final decision of the tree. For example, the first leaf node in the tree is reached when the value of "Department_gynecology" is less than or equal to 0.5, "Age_31-40" is less than or equal to 0.5, and "Age_41-50" is less than or equal to 0.5.

Interpretation and Conclusions:

- The decision tree starts with a split on the Department_gynecology feature. If the patient was not admitted to the gynecology department, the tree proceeds to consider the patient's age and the department of anesthesia.
    - If the patient is between 31-40 years old and was not admitted to the department of anesthesia, the tree reaches a leaf node and the predicted LOS for the patient is ~7 days.
    - If the patient is not admitted to the department of gynecology and is not between 31-50 years old, then the predicted LOS for the patient is ~27 days.
- If the patient was admitted to the gynecology department, the tree proceeds to consider the number of available extra rooms in the hospital and the type of admission.
    - If the patient was admitted with trauma and the number of available extra rooms is greater than 12.5, the tree reaches a leaf node and the predicted LOS for the patient is ~11 days.
    - If the number of available extra rooms is less than or equal to 12.5 and admission deposit is less than or equal to 4605.06, the tree reaches a leaf node and the predicted LOS for the patient is ~9 days.

Note: The tree is truncated and not shown completely to get proper visualization. You can try to plot the complete tree and get some more observations.

Bagging Regressor

What is a Bagging Regressor?

Bagging (short for Bootstrap Aggregating) is an ensemble learning technique that involves training multiple models on different subsets of the training data and then combining their predictions. The idea is to reduce variance and overfitting by averaging the predictions of many models.

A Bagging Regressor is a type of Bagging algorithm used for regression tasks. It involves training multiple regression models (e.g., Decision Trees) on different subsets of the training data and then combining their predictions by taking the average.

How does a Bagging Regressor work?

The Bagging Regressor works by generating multiple subsets of the training data by randomly selecting data points with replacement (i.e., allowing the same data point to be selected more than once in the same subset). Each subset is used to train a separate regression model, and the predictions of these models are combined by taking their average.

The idea behind this approach is that by training multiple models on different subsets of the data, we can reduce the variance and overfitting of the final model, while still

maintaining the same bias as a single model trained on the entire dataset.

$$Prediction = average\ of\ predictions\ of\ individual\ decision\ tree\ reg$$

In [39]:
```
# Bagging Regressor
bagging_estimator = BaggingRegressor(random_state = 1)

# Fitting the model
bagging_estimator.fit(x_train, y_train)

# Model Performance on the test data
bagging_estimator_perf_test = model_performance_regression(bagging_estimator

bagging_estimator_perf_test
```

Out[39]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	1.364505	0.902326	0.970434	0.970422	7.627444

Random Forest Regressor

What is a Random Forest?

Random Forest is another ensemble learning technique that combines multiple Decision Trees to create a more robust and accurate model. Like Bagging, it involves training multiple models on different subsets of the training data, but with an additional twist: at each split in the tree, only a random subset of the available features is considered for splitting.

This helps to reduce the correlation between the trees in the forest and improves their overall accuracy.

How does a Random Forest work?

A Random Forest works by training multiple Decision Trees on different subsets of the training data, and then combining their predictions by taking their average. The key difference from Bagging is that at each split in the tree, only a random subset of the features is considered for splitting.

The algorithm works as follows:

1. Generate multiple random subsets of the training data (with replacement).
2. For each subset, train a Decision Tree on a random subset of the features.
3. Make predictions for new data by averaging the predictions of all the trees in the forest.

The number of trees in the forest and the number of features considered at each split are hyperparameters that can be tuned to optimize the performance of the model.

The Random Forest algorithm doesn't have any specific equations, but it involves training multiple Decision Trees on different subsets of the training data with a random subset of the features considered at each split.

In [40]:
```
# Random Forest Regressor
rf_regressor = RandomForestRegressor(n_estimators = 100, random_state = 1)

# Fitting the model
rf_regressor.fit(x_train, y_train)

# Model Performance on the test data
rf_regressor_perf_test = model_performance_regression(rf_regressor, x_test,

rf_regressor_perf_test
```

Out[40]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	1.302336	0.863677	0.973067	0.973056	7.306138

AdaBoost

What is Adaboost?

Adaboost (short for Adaptive Boosting) is a type of boosting algorithm that combines multiple weak classifiers to create a stronger classifier. A weak classifier is a classifier that performs only slightly better than random guessing.

How does Adaboost work?

Adaboost works by training multiple weak classifiers on different subsets of the training data, and then combining their predictions to make a final prediction. The algorithm works as follows:

Assign equal weights to all the training examples. Train a weak classifier on a subset of the training data. Increase the weights of the misclassified examples. Train another weak classifier on the same subset of data but with the weights adjusted to give more importance to the misclassified examples. Repeat steps 3-4 for a specified number of iterations or until the error rate is sufficiently low. Combine the predictions of all the weak classifiers to make a final prediction. The key idea behind Adaboost is that by giving more weight to the misclassified examples, the algorithm can focus on the examples that are more difficult to classify and improve its overall accuracy.

The Adaboost algorithm involves computing the weighted error rate of each weak classifier and using it to update the weights of the training examples. The equation for computing the weighted error rate is:

$$\epsilon_t = \frac{\sum_{i=1}^{N} w_{t,i} \cdot \mathbb{I}(y_i \neq h_t(x_i))}{\sum_{i=1}^{N} w_{t,i}}$$

Here, w_i is the weight of the i_{th} training example, y_i is the true label of the i_{th} example, $h(x_i)$ is the prediction of the weak classifier for the i_{th} example, and the sum is over all the training examples.

The weight of the weak classifier is then computed as:

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

where α_t is the weight of the t_{th} weak learner in the final model, and ϵ_t is the weighted error of the t_{th} weak learner.

Finally, the weights of the training examples are updated as follows:

$$w_i \leftarrow w_i \exp(-\alpha y_i h(x_i))$$

Here, $exp()$ is the exponential function, y_i is the true label of the i_{th} example, $h(x_i)$ is the prediction of the weak classifier for the i_{th} example, and the sum is over all the training examples.

In [41]:
```
# Importing AdaBoost Regressor
from sklearn.ensemble import AdaBoostRegressor

# AdaBoost Regressor
ada_regressor = AdaBoostRegressor(random_state=1)

# Fitting the model
ada_regressor.fit(x_train, y_train)

# Model Performance on the test data
ada_regressor_perf_test = model_performance_regression(ada_regressor, x_test

ada_regressor_perf_test
```

Out[41]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	2.375388	1.58689	0.910399	0.910363	13.623722

Gradient Boosting Regressor

What is Gradient Boosting?

Gradient Boosting is another boosting algorithm that combines multiple weak learners to create a strong learner. The difference between Adaboost and Gradient Boosting is that

the former assigns different weights to different data points, while the latter fits the model to the residual errors of the previous model.

How does Gradient Boosting work?

Gradient Boosting works by sequentially adding weak learners to the model and updating the weights of the training examples based on the residual errors of the previous models. The algorithm works as follows:

Initialize the model with a constant value, such as the mean of the target variable.

For each weak learner:

- Train the weak learner on the training data.
- Compute the predictions of the weak learner.
- Compute the residual errors of the previous model by subtracting the predicted values from the actual values.
- Fit the weak learner to the residual errors.
- Update the weights of the training examples based on the fitted residual errors.
- Combine the predictions of all the weak learners to make a final prediction.

The key idea behind Gradient Boosting is that by fitting the model to the residual errors of the previous model, it can focus on the examples that were not well predicted by the previous model and improve its overall accuracy.

The Gradient Boosting algorithm involves computing the negative gradient of the loss function with respect to the predicted values and using it to update the model. The equation for computing the negative gradient is:

$$Negative\ Gradient = -\frac{\partial L(y_{\text{true}}, y_{\text{pred}})}{\partial y_{\text{pred}}}$$

Here, y_true is the true label of the example, y_pred is the predicted value of the model, and the partial derivatives are taken with respect to these variables.

The weight of the weak learner is then computed as:

$$\alpha = learning\ rate * negative\ gradient$$

Finally, the model is updated as:

$$model\ prediction = model\ prediction + alpha * weak\ learner\ predic$$

Here, learning_rate is a hyperparameter that controls the step size of each update, weak learner prediction is the prediction of the weak learner for the example, and the sum is over all the weak learners.

In [42]:
```
# Importing Gradient Boosting Regressor
from sklearn.ensemble import GradientBoostingRegressor

# Gradient Boosting Regressor
grad_regressor = GradientBoostingRegressor(random_state=1)

# Fitting the model
grad_regressor.fit(x_train, y_train)

# Model Performance on the test data
grad_regressor_perf_test = model_performance_regression(grad_regressor, x_te

grad_regressor_perf_test
```

Out[42]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	1.792721	1.212749	0.948965	0.948944	10.247284

XGBoost Regressor

What is XGBoost?

XGBoost (short for Extreme Gradient Boosting) is a highly optimized implementation of the Gradient Boosting algorithm. It was developed by Tianqi Chen at the University of Washington and is widely used in data science competitions.

How does XGBoost work?

XGBoost works by sequentially adding weak learners to the model and updating the weights of the training examples based on the residual errors of the previous models. The algorithm is similar to Gradient Boosting, but includes several additional features to improve its performance:

Regularization: XGBoost includes L1 and L2 regularization to prevent overfitting. **Tree Pruning**: XGBoost includes a technique called "tree pruning" to remove irrelevant features and reduce the complexity of the model. Weighted Quantile Sketch: XGBoost uses a weighted quantile sketch algorithm to speed up the computation of split points in the decision trees. Equations

The XGBoost algorithm involves computing the negative gradient of the loss function with respect to the predicted values and using it to update the model. The equation for computing the negative gradient is the same as in Gradient Boosting.

The weight of the weak learner is then computed as:

$$\alpha = learning\ rate * negative\ gradient + 0.5 * (L_1\ regularization + 1$$

Finally, the model is updated.

In [43]:
```
# Installing the xgboost library using the 'pip' command
!pip install xgboost
```

```
Collecting xgboost
  Downloading xgboost-2.1.3-py3-none-macosx_12_0_arm64.whl.metadata (2.1 kB)
Requirement already satisfied: numpy in /Users/obaozai/miniconda3/envs/jupyt
er/lib/python3.11/site-packages (from xgboost) (1.26.4)
Requirement already satisfied: scipy in /Users/obaozai/miniconda3/envs/jupyt
er/lib/python3.11/site-packages (from xgboost) (1.14.1)
Downloading xgboost-2.1.3-py3-none-macosx_12_0_arm64.whl (1.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 14.8 MB/s eta 0:00:00
a 0:00:01
Installing collected packages: xgboost
Successfully installed xgboost-2.1.3
```

In [44]:
```
# Importing XGBoost Regressor
from xgboost import XGBRegressor

# XGBoost Regressor
xgb = XGBRegressor(random_state = 1)

# Fitting the model
xgb.fit(x_train,y_train)

# Model Performance on the test data
xgb_perf_test = model_performance_regression(xgb, x_test, y_test)

xgb_perf_test
```

Out[44]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	1.513463	1.034136	0.963626	0.963612	8.868662

Models' Performance Comparison

Comparing different machine learning models is an important step in the modeling process, as it allows us to understand the strengths and weaknesses of each model, and to choose the best one for a particular task.

In the context of regression, we can compare models based on various performance metrics, such as **Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared, Adjusted R-squared and others**.

In [45]:
```
models_test_comp_df = pd.concat(
    [
        dt_regressor_perf_test.T,
        bagging_estimator_perf_test.T,
        rf_regressor_perf_test.T,
        ada_regressor_perf_test.T,
        grad_regressor_perf_test.T,
        xgb_perf_test.T
    ],
```

```
    axis = 1,
)

models_test_comp_df.columns = [
    "Decision tree regressor",
    "Bagging Regressor",
    "Random Forest regressor",
    "Ada Boost Regressor",
    "Gradient Boosting Regressor",
    "XG Boost Regressor"]

print("Test performance comparison:")

models_test_comp_df.T
```

```
Test performance comparison:
```

Out[45]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
Decision tree regressor	1.821321	1.131270	0.947324	0.947302	9.353216
Bagging Regressor	1.364505	0.902326	0.970434	0.970422	7.627444
Random Forest regressor	1.302336	0.863677	0.973067	0.973056	7.306138
Ada Boost Regressor	2.375388	1.586890	0.910399	0.910363	13.623722
Gradient Boosting Regressor	1.792721	1.212749	0.948965	0.948944	10.247284
XG Boost Regressor	1.513463	1.034136	0.963626	0.963612	8.868662

Choosing the Models for Tuning Hyperparameters

Choosing the final model from the set of compared models depends on various factors. Here are some steps to help you make a decision:

- **Look at the evaluation metrics**: Check the evaluation metrics of the models that you have compared. Choose the model that performs the best based on your criteria. However, it is important to keep in mind that the model with the best performance on the training set may not necessarily be the best on the test set. Therefore, it is important to also consider the model's performance on the test set.
- **Overfitting**: Check for overfitting in the models. A model that overfits the data may perform very well on the training set but poorly on the test set. One way to check for overfitting is by comparing the performance of the model on the training set and the test set. If the difference in performance is large, it may indicate overfitting. Therefore, it is better to choose a model that has a good balance between performance on the training set and the test set.

- **Model complexity**: Consider the complexity of the models. A more complex model may fit the data better but may also overfit the data. Therefore, it is better to choose a model that has a good balance between simplicity and performance.
- **Interpretability**: Consider the interpretability of the models. Some models, such as decision trees, are more interpretable than others, such as neural networks. If interpretability is important for your application, it may be better to choose a more interpretable model.
- **Runtime**: Consider the runtime of the models. Some models may take longer to train and predict than others. If runtime is a concern, it may be better to choose a model that is faster to train and predict.

Overall, the choice of the final model should be based on a combination of the above factors, as well as the specific requirements and constraints of your application. Hence, there are are no strict rules of choosing the best model. It depends on the dataset and the business problem at hand.

Observations:

- Based on the results obtained after comparing all of the models, the **Random Forest Regressor** is the best-performing model.
- The **Random Forest Regressor** has the **lowest RMSE and MAE**, indicating that the average difference between predicted and actual values is the smallest. It also has a **higher R-squared and Adjusted R-squared**, indicating that the model explains a significant proportion of the variance in the target variable. It also has a **low MAPE**, indicating that it has a small average percentage error.
- Because the Random Forest model performs well on test data, it is not overfitting the training data. Random Forest is also less complex than boosting models such as XGBoost.
- The Random Forest has a longer runtime in comparison to other models like Decision Tree. Hence, there is a trade-off between runtime and model performance. In this case, we are prioritizing the model performance over runtime, but other approaches are possible depending on the scenario.
- Let's see if we can improve the model performance by **tuning the hyperparameters** of the Random Forest model. Hyperparameter tuning is a crucial step in machine learning as it helps **to optimize the model's performance** by finding the best set of hyperparameters that work well for the given dataset.

Tuning the Model

Tuning the hyperparameters of a machine learning model can help improve its performance. Here are some steps you can follow to tune the hyperparameters of your model:

- **Identify the hyperparameters**: Before tuning the hyperparameters, it's important to identify the hyperparameters that can be tuned. In the case of the models you have built (Decision Trees, Bagging Regressor, Random Forest, AdaBoost, Gradient Boosting, XGBoost), some of the hyperparameters that can be tuned include the number of estimators, learning rate, maximum depth, minimum sample split, etc.
- **Determine the range of values for each hyperparameter**: Once you have identified the hyperparameters, you need to determine the range of values that each hyperparameter can take. For example, you can set the range for the number of estimators to be between 50 and 200.
- **Choose a method to search for the best hyperparameters**: There are different methods for searching for the best hyperparameters, such as grid search and randomized search. **Grid search** is a simple and exhaustive method that involves evaluating the model performance for all possible combinations of hyperparameters within the specified range. **Randomized search** is similar to grid search, but instead of evaluating all possible combinations, it evaluates a random subset of combinations.
- **Train and evaluate the model with each combination of hyperparameters**: Once you have chosen a method to search for the best hyperparameters, you need to train and evaluate the model with each combination of hyperparameters within the specified range.
- **Select the hyperparameters that give the best performance**: Finally, you need to select the hyperparameters that give the best performance on the validation set. You can then use these hyperparameters to train the model on the full training set and evaluate its performance on the test set.

Overall, tuning the hyperparameters of a model can be a time-consuming process, but it can greatly improve the performance of the model.

Tuned Random Forest Regressor

Note: Depending on the size of the dataset, the number of hyperparameters passed, the number of values passed for each hyperparameter, and the system's configuration, running the code below may take some time.

In [ ]:
```
rf_tuned = RandomForestRegressor(random_state = 1)

# Grid of parameters to choose from
```

```
rf_parameters = {"n_estimators": [100, 110, 120],

    "max_depth": [5, 7, None],

    "max_features": [0.8, 1]
             }

# Run the grid search
rf_grid_obj = GridSearchCV(rf_tuned, rf_parameters, scoring = 'neg_mean_squa

rf_grid_obj = rf_grid_obj.fit(x_train, y_train)

# Set the rf_tuned_regressor to the best combination of parameters
rf_tuned_regressor = rf_grid_obj.best_estimator_

rf_tuned_regressor.fit(x_train, y_train)

# Model Performance on the test data
rf_tuned_regressor_perf_test = model_performance_regression(rf_tuned_regress

rf_tuned_regressor_perf_test
```

Choosing the Final Model

In [ ]:
```
models_test_comp_df = pd.concat(
    [
        dt_regressor_perf_test.T,
        bagging_estimator_perf_test.T,
        rf_regressor_perf_test.T,
        ada_regressor_perf_test.T,
        grad_regressor_perf_test.T,
        xgb_perf_test.T,
        rf_tuned_regressor_perf_test.T,
    ],
    axis = 1,
)

models_test_comp_df.columns = [
    "Decision tree regressor",
    "Bagging Regressor",
    "Random Forest regressor",
    "Ada Boost Regressor",
    "Gradient Boosting Regressor",
    "XG Boost Regressor",
    "Random Forest Tuned Regressor"]

print("Test performance comparison:")

models_test_comp_df.T
```

Observations:

- After tuning, the performance of **Random Forest Tuned** model has slightly **improved in terms of RMSE and R-squared values**, as compared to the model

with default value of the hyperparameters. Hence, we can choose the Random Forest Tuned model as the final model.

Visualizing the Feature Importance

In [ ]:
```
# Plotting the feature importance
features = list(x.columns)

importances = rf_tuned_regressor.feature_importances_

indices = np.argsort(importances)

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

plt.title('Feature Importances')

plt.barh(range(len(indices)), importances[indices], color = 'violet', align

plt.yticks(range(len(indices)), [features[i] for i in indices])

plt.xlabel('Relative Importance')

plt.show()
```

Observations:

- **The most important features** are Department_gynecology, Age_41_50, and Age_31_40, followed by Department_anesthesia, Department_radiotherapy, and Admission_Deposit.
- The rest of the variables have little or no influence on the length of stay in the hospital in this model.

Business Insights and Recommendations

- Gynecology is the busiest department of the hospital and it handles 68.7% of the total number of patients. It needs ample resources and staff for the smooth functioning of the department.
- The maximum number of visitors can go up to 32 which is very high. A restriction can be imposed on this.
- 74.2% of the patients are female. Thus, resources need to be procured while keeping this figure in mind.
- A large percentage of patients (89.3%) are in trauma or emergency during admission. An increase in ambulances and emergency rooms can reduce the risk of casualties.
- Ward A has the most number of patients who stay for the longest and the most serious patients. These wards can be equipped with more resources and staff to

reduce the length of stay of these patients.

- Elderly patients (51-100) and children (1-10) stay for the longest. Extra attention to these age groups can lead to a faster discharge from the hospital.
- Wards D, E, and C have the most visitors with a patient. These wards will need more space and amenities like washrooms, shops, and lobbies for the visitors. Spaces can also be rented out to shop owners and advertisements to generate extra income.
- Finally, the Random Forest Regressor can predict the length of stay of the patient with just an error of 1 day. The hospital can use these predictions to allocate the resources and staff accordingly and reduce any kind of wastage. The hospital can also allocate the wards and doctors accordingly to optimize admissions even during emergencies.

Next Steps

- The next step is creating a pipeline that includes a Column Transformer to preprocess the data and a model that has been trained on the data. This pipeline can be used in future applications or as a starting point for further model development.

- Using a pipeline with a Column Transformer is a common practice in machine learning to ensure that data preprocessing is consistent and can be easily reproduced. The pipeline will take care of data preprocessing and model training in a single step, making it easy to use the model in other applications.

- Saving the trained model in the Pickle format allows for easy serialization and deserialization of the model, making it possible to use the model in other applications without needing to retrain it. This is particularly useful when working with large datasets or models that take a long time to train.