1. Downlaoding/Uploading Data

In [3]:
```
!git clone https://github.com/ciol-researchlab/CIOL-Winter-ML-Bootcamp.git
```

```
fatal: destination path 'CIOL-Winter-ML-Bootcamp' already exists and is not an empty directory.
```

2. Setting up the enviroment

In [4]:
```
# Tabjular Data Analysis
import numpy as np
import pandas as pd

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Utility
import time
import warnings
warnings.filterwarnings('ignore')
```

In [31]:
```
cls_df = pd.read_csv("/content/CIOL-Winter-ML-Bootcamp/datasets/session2/main/spaceship-titanic/data.csv")
cls_df[:500]
cls_df.head()
```

Out[31]:

	PassengerId	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	
0	0001_01	Europa	False	B/0/P	TRAPPIST-1e	39.0	False	0.0	0.0	0.0	0.0	0.0	
1	0002_01	Earth	False	F/0/S	TRAPPIST-1e	24.0	False	109.0	9.0	25.0	549.0	44.0	
2	0003_01	Europa	False	A/0/S	TRAPPIST-1e	58.0	True	43.0	3576.0	0.0	6715.0	49.0	
3	0003_02	Europa	False	A/0/S	TRAPPIST-1e	33.0	False	0.0	1283.0	371.0	3329.0	193.0	
4	0004_01	Earth	False	F/1/S	TRAPPIST-1e	16.0	False	303.0	70.0	151.0	565.0	2.0	Sa

In [32]:
```
reg_df = pd.read_csv("/content/CIOL-Winter-ML-Bootcamp/datasets/session2/main/house-price/data.csv")
reg_df[:500]
reg_df.head()
```

Out[32]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	I
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	

5 rows × 81 columns

In [33]:
```
m_cls_df = pd.read_csv("/content/CIOL-Winter-ML-Bootcamp/datasets/session2/main/iris/data.csv")
m_cls_df.head()
```

Out[33]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

3. Data Pre-processing

In [8]:
```
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder, OneHotEncoder, StandardScaler
from sklearn.model_selection import StratifiedKFold, train_test_split
```

In [9]:
```
# Dummy Variable
dfx = pd.DataFrame()
```

Drop unnecessary data

In [34]:
```
cls_df.shape
```

Out[34]:
```
(8693, 14)
```

In [35]:
```
cls_df.head()
```

Out[35]:

	PassengerId	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	
0	0001_01	Europa	False	B/0/P	TRAPPIST-1e	39.0	False	0.0	0.0	0.0	0.0	0.0	
1	0002_01	Earth	False	F/0/S	TRAPPIST-1e	24.0	False	109.0	9.0	25.0	549.0	44.0	
2	0003_01	Europa	False	A/0/S	TRAPPIST-1e	58.0	True	43.0	3576.0	0.0	6715.0	49.0	
3	0003_02	Europa	False	A/0/S	TRAPPIST-1e	33.0	False	0.0	1283.0	371.0	3329.0	193.0	
4	0004_01	Earth	False	F/1/S	TRAPPIST-1e	16.0	False	303.0	70.0	151.0	565.0	2.0	Sa

In [36]:
```
columns_to_drop = ['PassengerId', 'Name']
cls_df = cls_df.drop(columns=columns_to_drop)
cls_df.shape
```

Out[36]:
```
(8693, 12)
```

Train-test split

What is a Train Set and Test Set?

1. **Train Set**:
   - The **train set** is a subset of your dataset used to train your machine learning model.
   - It is used by the model to learn the patterns, relationships, and structures in the data to make predictions.
2. **Test Set**:
   - The **test set** is a separate subset of the dataset used to evaluate the performance of the trained model.
   - It contains data that the model has never seen during training, allowing you to assess how well the model generalizes to new, unseen data.

Why are Train and Test Sets Necessary?

1. **Model Evaluation**:
   - **Overfitting Prevention**: If the model is trained and tested on the same data, it may memorize the training data (overfitting), and thus, it would perform poorly on new, unseen data. Splitting the data into train and test sets helps avoid this issue.
   - **Generalization Check**: The test set acts as a proxy for real-world, unseen data. Evaluating on the test set helps check how well the model generalizes its learning.
2. **Performance Assessment**:
   - The test set provides a fair way to assess the performance of your model. Without a test set, you can't reliably tell how well your model will perform in production or on new data.
3. **Model Tuning**:
   - When tuning hyperparameters (like in cross-validation), the train/test split ensures that the validation and test processes remain independent, preventing bias.

Typical Split Ratio:

- A common split is **80% for training and 20% for testing**, but this can vary based on the dataset size and the specific use case.
- In some cases, especially with very large datasets, a **70/30** or even **90/10** split can be used.

In [12]:
```
train_cls_df, test_cls_df = train_test_split(cls_df, test_size=0.2, random_state=42)
```

In [37]:
```
train_cls_df.shape
```

Out[37]:
```
(6954, 12)
```

In [38]:
```
test_cls_df.shape
```

Out[38]:
```
(1739, 12)
```

Handle Missing Data

In [39]:
```
# Select numerical columns
numerical_columns = cls_df.select_dtypes(include=['number']).columns.tolist()
print("Numerical Columns:", numerical_columns)
```

```
Numerical Columns: ['Age', 'RoomService', 'FoodCourt', 'ShoppingMall', 'Spa', 'VRDeck']
```

In [40]:
```
# Select categorical columns
categorical_columns = cls_df.select_dtypes(include=['object', 'category']).columns.tolist()
print("Categorical Columns:", categorical_columns)
```

```
Categorical Columns: ['HomePlanet', 'CryoSleep', 'Cabin', 'Destination', 'VIP']
```

In [17]:
```
columns_with_missing = cls_df.columns[cls_df.isnull().any()].tolist()
print("Missing data Columns:", columns_with_missing)
```

```
Missing data Columns: ['HomePlanet', 'CryoSleep', 'Cabin', 'Destination', 'Age', 'VIP', 'RoomService', 'FoodCour
t', 'ShoppingMall', 'Spa', 'VRDeck']
```

`Note: imputer only works with numerical columns`

So, let's find all numerical missing columns.

In [18]:
```
imputer_cols = ["Age", "FoodCourt", "ShoppingMall", "Spa", "VRDeck" ,"RoomService"]
```

In [19]:
```
imputer = SimpleImputer(strategy='median')
imputer.fit(train_cls_df[imputer_cols])
```

Out[19]:
```
SimpleImputer
SimpleImputer(strategy='median')
```

In [20]:
```
train_cls_df[imputer_cols] = imputer.transform(train_cls_df[imputer_cols])
test_cls_df[imputer_cols] = imputer.transform(test_cls_df[imputer_cols])
```

In [41]:
```
train_cls_df.columns[train_cls_df.isnull().any()].tolist()
```

Out[41]:
```
['Cabin']
```

In [42]:
```
train_cls_df.head()
```

Out[42]:

	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Transported
2333	0	0	NaN	2	28.0	0	0.0	55.0	0.0	656.0	0.0	False
2589	0	0	F/575/P	2	17.0	0	0.0	1195.0	31.0	0.0	0.0	False
8302	1	1	C/329/S	0	28.0	0	0.0	0.0	0.0	0.0	0.0	True
8177	2	0	F/1800/P	2	20.0	0	0.0	2.0	289.0	976.0	0.0	True
500	1	1	C/18/P	0	36.0	0	0.0	0.0	0.0	0.0	0.0	True

Categorical Variables

1. Label Encoding

Definition: Converts categorical values into integer labels. Each unique value in the column is assigned a numerical label.

Example:

HomePlanet	Encoded
Earth	0
Mars	1
Venus	2

In [43]:
```
train_cls_df['HomePlanet'].head()
```

Out[43]:

	HomePlanet
2333	0
2589	0
8302	1
8177	2
500	1

dtype: int64

In [22]:
```
# Apply Label Encoding
label_encoder = LabelEncoder()
dfx['HomePlanet_label'] = label_encoder.fit_transform(train_cls_df['HomePlanet'])
dfx['HomePlanet_label'].head()
```

Out[22]:

	HomePlanet_label
0	0
1	0
2	1
3	2
4	1

dtype: int64

---

2. Ordinal Encoding

Definition: Converts categorical values into integers, preserving order if the categories have a natural rank. If no natural order exists, it behaves similarly to label encoding.

Example: Assuming "Mars" > "Earth" > "Venus":

HomePlanet	Encoded
Earth	1
Mars	2
Venus	0

In [23]:
```
# Apply Ordinal Encoding
ordinal_encoder = OrdinalEncoder()
dfx['CryoSleep_label'] = ordinal_encoder.fit_transform(train_cls_df[['CryoSleep']])
dfx['CryoSleep_label'].head()
```

Out[23]:

	CryoSleep_label
0	0.0
1	0.0
2	1.0
3	0.0
4	1.0

dtype: float64

---

3. One-Hot Encoding

Definition: Represents each category as a binary vector. Each unique value becomes a new column, and rows have `1` in the column corresponding to their category and `0` elsewhere.

Example:

HomePlanet	Earth	Mars	Venus
Earth	1	0	0
Mars	0	1	0
Venus	0	0	1

In [24]:
```
# Apply One-Hot Encoding
encoder = OneHotEncoder(sparse_output=False)

# Fit and transform the categorical columns
one_hot_encoded = encoder.fit_transform(train_cls_df[['HomePlanet']])

# Create a DataFrame with the encoded columns
one_hot_df = pd.DataFrame(one_hot_encoded,
                          columns=encoder.get_feature_names_out(['HomePlanet']))
one_hot_df.head()
```

Out[24]:

	HomePlanet_Earth	HomePlanet_Europa	HomePlanet_Mars	HomePlanet_nan
0	1.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0
3	0.0	0.0	1.0	0.0
4	0.0	1.0	0.0	0.0

Let's apply

In [44]:
```
train_cls_df[categorical_columns].head(5)
```

Out[44]:

	HomePlanet	CryoSleep	Cabin	Destination	VIP
2333	0	0	NaN	2	0
2589	0	0	F/575/P	2	0
8302	1	1	C/329/S	0	0
8177	2	0	F/1800/P	2	0
500	1	1	C/18/P	0	0

In [26]:
```
train_cls_df[categorical_columns].nunique()
```

Out[26]:

	0
HomePlanet	3
CryoSleep	2
Cabin	5441
Destination	3
VIP	2

dtype: int64

In [27]:
```
apply_category_encoding = ['HomePlanet', 'CryoSleep', 'Destination', 'VIP']
```

In [28]:
```
def label_encoder(train, test, columns):
    """
    Encodes categorical features in both training and testing datasets using Label Encoding.

    Parameters:
    - train: DataFrame containing the training data.
    - test: DataFrame containing the testing data.
    - columns: List of column names (categorical features) to encode.

    Returns:
    - train: Transformed training DataFrame with encoded categorical columns.
    - test: Transformed testing DataFrame with encoded categorical columns.
    """
    encoder = LabelEncoder()

    for col in columns:
        # Convert column values to strings to handle mixed data types
        train[col] = train[col].astype(str)
        test[col] = test[col].astype(str)

        # Apply Label Encoding to convert categorical values into numerical labels
        train[col] = encoder.fit_transform(train[col])  # Encodes the training column
        test[col] = encoder.transform(test[col])    # Encodes the testing column

    return train, test
```

In [29]:
```
train_cls_df ,test_cls_df = label_encoder(train_cls_df,test_cls_df , apply_category_encoding)
```

4. Feature Engineering

What is Feature Engineering?

Feature engineering is the process of creating, modifying, and selecting features (input variables) from raw data to improve the performance of machine learning models. It involves techniques such as:

1. **Feature Transformation**: Transforming raw data into more meaningful representations (e.g., scaling, encoding).
2. **Feature Extraction**: Creating new features from existing ones (e.g., extracting day-of-week from a timestamp).
3. **Feature Selection**: Identifying the most relevant features to reduce dimensionality and improve model efficiency.

Why is Feature Engineering Important?

1. **Improves Model Performance**:

- High-quality features can significantly boost the accuracy and effectiveness of a machine learning model.
- Poorly chosen or irrelevant features can lead to underperforming models.

2. **Reduces Overfitting**:
   - By selecting only the most relevant features, the model is less likely to overfit the training data.
3. **Handles Data Complexity**:
   - Real-world data is often messy, incomplete, and unstructured. Feature engineering can clean and transform such data into usable formats.
4. **Enhances Interpretability**:
   - Well-designed features make models easier to interpret, enabling better insights from data.
5. **Optimizes Computation**:
   - Reducing the number of irrelevant features improves computational efficiency and reduces training time.
6. **Adapts to Specific Domain Needs**:
   - Domain knowledge helps create features that are more relevant and tailored to the problem at hand (e.g., using interaction terms in economics or deriving chemical properties in molecular ML).

Example 1: Feture Extraction

In [45]:
```
train_cls_df["Cabin"].head()
```

Out[45]:

	Cabin
2333	NaN
2589	F/575/P
8302	C/329/S
8177	F/1800/P
500	C/18/P

dtype: object

In [46]:
```
"I have joined CIOL Winter ML Workshop".split()
```

Out[46]:
```
['I', 'have', 'joined', 'CIOL', 'Winter', 'ML', 'Workshop']
```

In [48]:
```
"I have joined CIOL Winter ML Workshop".split()[3]
```

Out[48]:
```
'CIOL'
```

In [50]:
```
"I have joined CIOL Winter ML Workshop".split("a")
```

Out[50]:
```
['I h', 've joined CIOL Winter ML Workshop']
```

In [ ]:
```
train_cls_df.head()
```

In [ ]:
```

```

In [51]:
```
def cabin_new_feature(df):
    df["Cabin"].fillna("np.nan/np.nan/np.nan",inplace=True)

    df["Cabin_Deck"] = df["Cabin"].apply(lambda x: x.split("/")[0])
    df["Cabin_Number"]  = df["Cabin"].apply(lambda x: x.split("/")[1])
    df["Cabin_Side"] = df["Cabin"].apply(lambda x: x.split("/")[2])

    #Replacing string nan values to numpy nan values..
    cols = ["Cabin_Deck","Cabin_Number","Cabin_Side"]
    df[cols] = df[cols].replace("np.nan",np.nan)

    #Filling Missing Values in new features created.
    df["Cabin_Deck"].fillna(df["Cabin_Deck"].mode()[0],inplace=True)
    df["Cabin_Side"].fillna(df["Cabin_Side"].mode()[0],inplace=True)
    df["Cabin_Number"].fillna(df["Cabin_Number"].mode()[0],inplace=True)

    return df
```

In [52]:
```
train_cls_df = cabin_new_feature(train_cls_df)
test_cls_df = cabin_new_feature(test_cls_df)
```

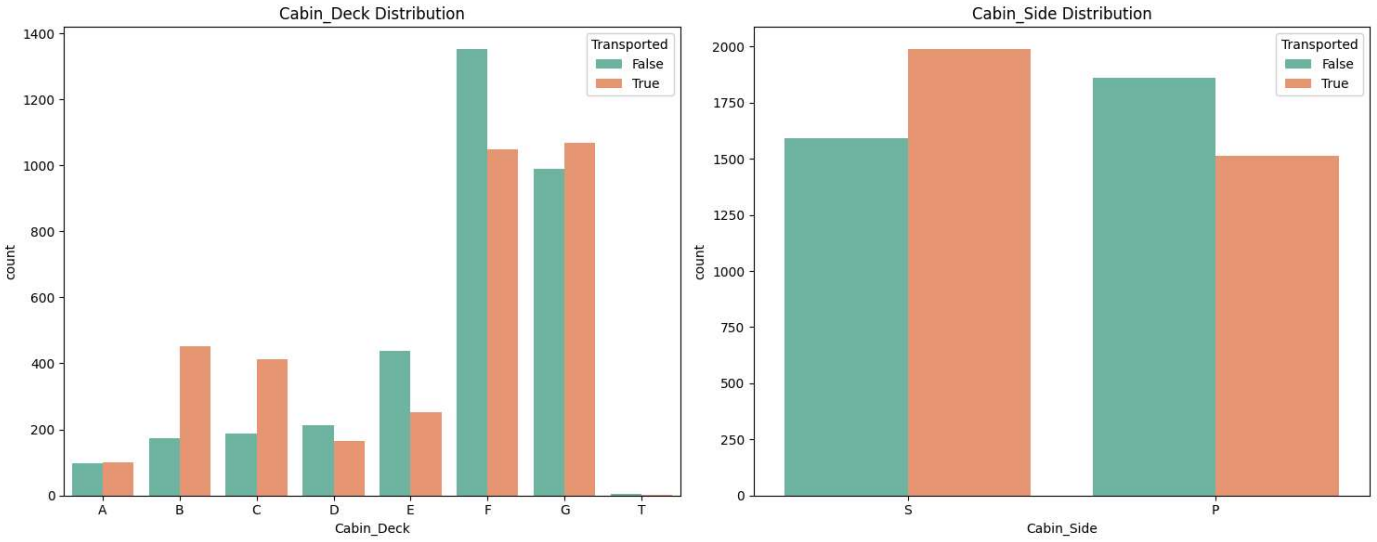
In [53]:
```
train_cls_df.head()
```

Out[53]:

	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck
2333	0	0	np.nan/np.nan/np.nan	2	28.0	0	0.0	55.0	0.0	656.0	0.0
2589	0	0	F/575/P	2	17.0	0	0.0	1195.0	31.0	0.0	0.0
8302	1	1	C/329/S	0	28.0	0	0.0	0.0	0.0	0.0	0.0
8177	2	0	F/1800/P	2	20.0	0	0.0	2.0	289.0	976.0	0.0
500	1	1	C/18/P	0	36.0	0	0.0	0.0	0.0	0.0	0.0

In [54]:
```
plt.figure(figsize=(15,6))
plt.subplot(1,2,1)
sns.countplot(x="Cabin_Deck",hue="Transported", data=train_cls_df, palette="Set2",order=["A","B","C","D","E","F",
plt.title("Cabin_Deck Distribution")

plt.subplot(1,2,2)
sns.countplot(x="Cabin_Side", hue="Transported", data=train_cls_df, palette="Set2")
plt.title("Cabin_Side Distribution")
plt.tight_layout()
plt.show()
```



In [55]:
```
train_cls_df ,test_cls_df = label_encoder(train_cls_df,test_cls_df , ['Cabin_Deck','Cabin_Side'])
```

In [56]:
```
train_cls_df["Cabin_Number"] = train_cls_df["Cabin_Number"].astype(int)
test_cls_df["Cabin_Number"] = test_cls_df["Cabin_Number"].astype(int)
```

In [57]:
```
train_cls_df.head()
```

Out[57]:

	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck
2333	0	0	np.nan/np.nan/np.nan	2	28.0	0	0.0	55.0	0.0	656.0	0.0
2589	0	0	F/575/P	2	17.0	0	0.0	1195.0	31.0	0.0	0.0
8302	1	1	C/329/S	0	28.0	0	0.0	0.0	0.0	0.0	0.0
8177	2	0	F/1800/P	2	20.0	0	0.0	2.0	289.0	976.0	0.0
500	1	1	C/18/P	0	36.0	0	0.0	0.0	0.0	0.0	0.0

Example 2: Feture Creation

In [58]:
```
train_cls_df["Total Expenditure"] = train_cls_df['RoomService'] + train_cls_df['FoodCourt'] + train_cls_df['Shopp
test_cls_df["Total Expenditure"] = test_cls_df['RoomService'] + test_cls_df['FoodCourt'] + test_cls_df['ShoppingM
```

In [59]:
```
train_cls_df.head()
```

Out[59]:

	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck
2333	0	0	np.nan/np.nan/np.nan	2	28.0	0	0.0	55.0	0.0	656.0	0.0
2589	0	0	F/575/P	2	17.0	0	0.0	1195.0	31.0	0.0	0.0
8302	1	1	C/329/S	0	28.0	0	0.0	0.0	0.0	0.0	0.0
8177	2	0	F/1800/P	2	20.0	0	0.0	2.0	289.0	976.0	0.0
500	1	1	C/18/P	0	36.0	0	0.0	0.0	0.0	0.0	0.0

Example 3: Feature Transformation

In [60]:
```
train_cls_df.describe().T
```

Out[60]:

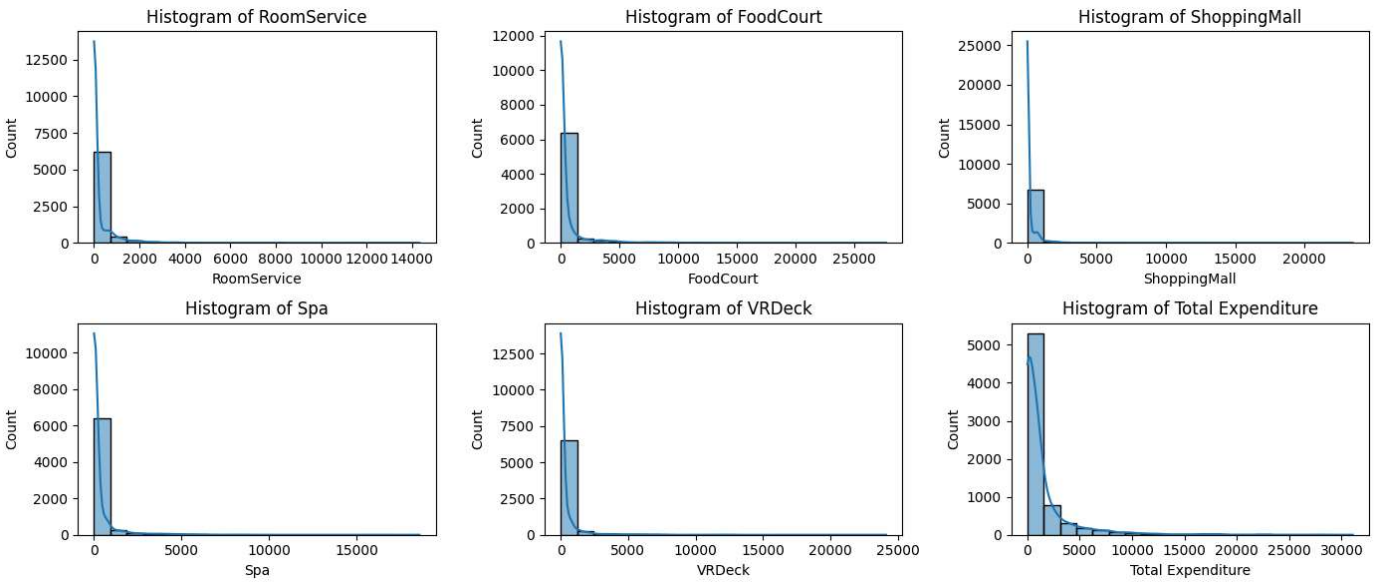
	count	mean	std	min	25%	50%	75%	max
HomePlanet	6954.0	0.717860	0.865003	0.0	0.0	0.0	1.00	3.0
CryoSleep	6954.0	0.396031	0.538645	0.0	0.0	0.0	1.00	2.0
Destination	6954.0	1.512367	0.840462	0.0	1.0	2.0	2.00	3.0
Age	6954.0	28.789186	14.294256	0.0	20.0	27.0	37.00	79.0
VIP	6954.0	0.069025	0.332969	0.0	0.0	0.0	0.00	2.0
RoomService	6954.0	218.785591	669.417469	0.0	0.0	0.0	41.75	14327.0
FoodCourt	6954.0	447.519988	1560.181622	0.0	0.0	0.0	76.00	27723.0
ShoppingMall	6954.0	171.334915	607.228282	0.0	0.0	0.0	23.00	23492.0
Spa	6954.0	309.610584	1108.098517	0.0	0.0	0.0	55.00	18572.0
VRDeck	6954.0	297.584556	1158.706646	0.0	0.0	0.0	41.00	24133.0
Cabin_Deck	6954.0	4.327150	1.759719	0.0	3.0	5.0	6.00	7.0
Cabin_Number	6954.0	591.069744	511.332454	0.0	151.0	416.5	982.75	1894.0
Cabin_Side	6954.0	0.514668	0.499821	0.0	0.0	1.0	1.00	1.0
Total Expenditure	6954.0	1444.835634	2795.443041	0.0	0.0	719.0	1436.50	31074.0

In [61]:
```
expenditure_columns = ['RoomService','FoodCourt','ShoppingMall','Spa','VRDeck','Total Expenditure']
```

In [62]:
```
# Set up the plotting area (adjust size as needed)
plt.figure(figsize=(14, 6))

# Plot each categorical column's value counts
for i, col in enumerate(expenditure_columns):
    plt.subplot(2, 3, i+1)  # Adjust grid size (3x3 here)
    sns.histplot(train_cls_df[col], bins=20, kde=True)
    plt.title(f'Histogram of {col}')

# Tight layout for better spacing
plt.tight_layout()
plt.show()
```

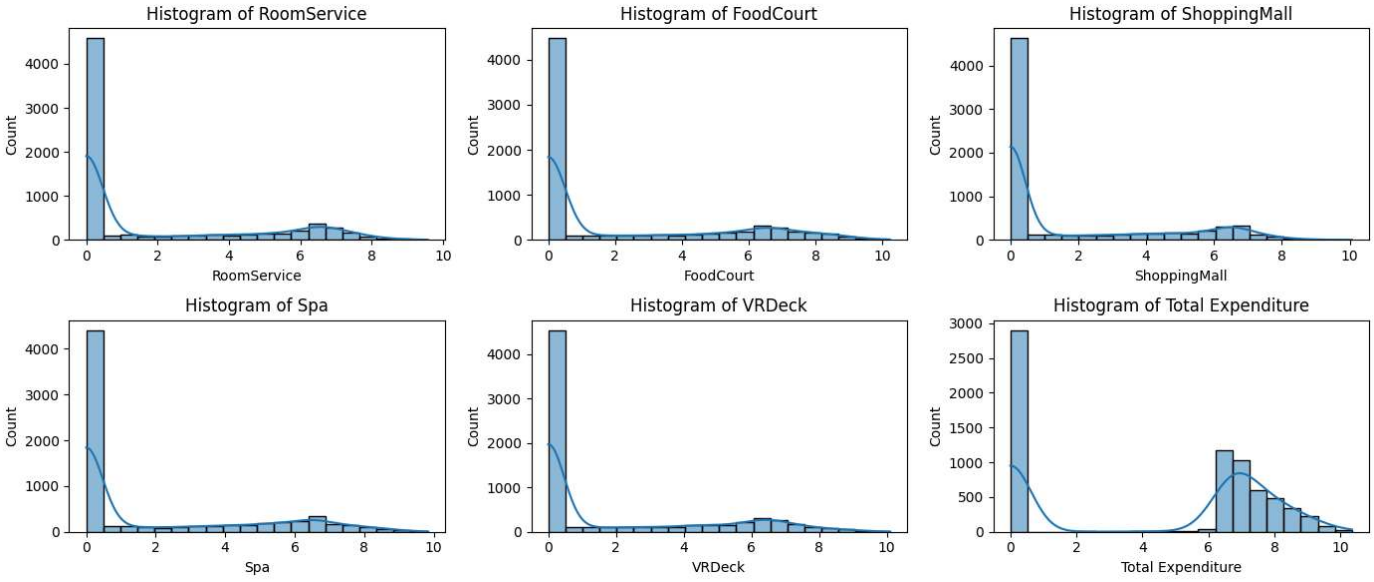


In [63]:
```
for col in expenditure_columns:
    train_cls_df[col] = np.log1p(train_cls_df[col])
    test_cls_df[col] = np.log1p(test_cls_df[col])
```

In [64]:
```
# Set up the plotting area (adjust size as needed)
plt.figure(figsize=(14, 6))

# Plot each categorical column's value counts
for i, col in enumerate(expenditure_columns):
    plt.subplot(2, 3, i+1)
    sns.histplot(train_cls_df[col], bins=20, kde=True)
    plt.title(f'Histogram of {col}')

# Tight layout for better spacing
plt.tight_layout()
plt.show()
```



In [65]:
```
# Convert True/False to 1/0
train_cls_df['Transported'] = train_cls_df['Transported'].astype(int)
test_cls_df['Transported'] = test_cls_df['Transported'].astype(int)
```

Example 4: Feature Selection

In [66]:
```
train_cls_df
```

Out[66]:

	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDe
2333	0	0	np.nan/np.nan/np.nan	2	28.0	0	0.000000	4.025352	0.000000	6.487684	0.0000
2589	0	0	F/575/P	2	17.0	0	0.000000	7.086738	3.465736	0.000000	0.0000
8302	1	1	C/329/S	0	28.0	0	0.000000	0.000000	0.000000	0.000000	0.0000
8177	2	0	F/1800/P	2	20.0	0	0.000000	1.098612	5.669881	6.884487	0.0000
500	1	1	C/18/P	0	36.0	0	0.000000	0.000000	0.000000	0.000000	0.0000
...	...	...	...	...	...	...	...	...	...	...	
5734	0	2	G/988/S	2	18.0	0	2.708050	1.098612	4.976734	6.415097	0.0000
5191	2	0	F/1063/S	2	50.0	2	6.538140	0.000000	3.433987	6.637258	6.0614
5390	0	0	F/1194/P	1	22.0	0	5.068904	0.000000	6.167516	0.000000	3.2958
860	2	0	F/191/P	2	34.0	0	5.940171	0.000000	7.394493	0.000000	0.0000
7270	1	0	C/253/P	0	28.0	0	2.079442	6.194405	0.000000	1.609438	8.7041

6954 rows × 16 columns

In [67]:
```
train_cls_df = train_cls_df.drop(columns=['Cabin'])
test_cls_df = test_cls_df.drop(columns=['Cabin'])
```

Final Data

In [68]:
```
train_cls_df.head()
```

Out[68]:

	HomePlanet	CryoSleep	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Transported	Cabi
2333	0	0	2	28.0	0	0.0	4.025352	0.000000	6.487684	0.0	0	
2589	0	0	2	17.0	0	0.0	7.086738	3.465736	0.000000	0.0	0	
8302	1	1	0	28.0	0	0.0	0.000000	0.000000	0.000000	0.0	1	
8177	2	0	2	20.0	0	0.0	1.098612	5.669881	6.884487	0.0	1	
500	1	1	0	36.0	0	0.0	0.000000	0.000000	0.000000	0.0	1	

In [69]:
```
test_cls_df.head()
```

Out[69]:

	HomePlanet	CryoSleep	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Transported	Cabi
304	2	0	2	19.0	0	6.035481	5.857933	6.453625	1.386294	6.964136	1	
2697	0	0	2	18.0	0	1.609438	6.807935	0.000000	0.000000	0.693147	0	
8424	0	1	2	41.0	0	0.000000	0.000000	0.000000	0.000000	0.000000	0	
1672	0	0	2	35.0	0	0.000000	5.826000	6.079933	0.000000	0.000000	1	
8458	1	1	2	43.0	0	0.000000	0.000000	0.000000	0.000000	0.000000	1	

We'll resume at 9:48 :)

5. Experimental Setup

What is a Holdout (Development) Set?

A **holdout set**, also known as a **development set**, is a subset of the data that is used during the model development process, but it is separate from both the training set and the final test set. It is used for tasks such as model tuning, hyperparameter optimization, and intermediate model evaluation.

Purpose of a Holdout/Development Set:

1. **Model Tuning and Hyperparameter Optimization**:
   - The holdout set is used to tune the model's hyperparameters or test different model configurations before final testing. It allows you to adjust the model based on performance without contaminating the final evaluation using the test set.
2. **Preventing Overfitting**:
   - By providing an additional validation stage during the model development, the holdout set helps prevent overfitting on the training data. It serves as a checkpoint to evaluate the model during the iterative process of model development.
3. **Simulating Real-World Scenarios**:
   - The holdout set serves as a mock of real-world data. It allows you to evaluate how well the model is performing on data that it hasn't been explicitly trained on, but without using the final test set.

Typical Data Split:

- **Training Set**: Used to train the model (typically 60-80% of the data).
- **Holdout/Development Set**: Used to tune the model and evaluate its performance during development (typically 10-20% of the data).
- **Test Set**: Used only once at the very end to assess the model's final performance and generalizability (typically 10-20% of the data).

In [70]:
```
X = test_cls_df.drop('Transported', axis =1 )
y = test_cls_df['Transported']
X_train , X_test , y_train , y_test = train_test_split(X , y, random_state = 12 , test_size = 0.25)
```

6. Classification Models

1. Binary Classification

Definition:

Binary classification is a type of classification problem where the target variable has two possible outcomes or classes. These two classes are typically labeled as `0` and `1`, `True` and `False`, or any other two distinct labels. The goal of binary classification is to predict which of the two classes a given instance belongs to based on the input features.

Example:

- **Spam Email Detection**: A model classifies emails into two categories: spam (1) or not spam (0).
- **Disease Diagnosis**: A model predicts whether a patient has a certain disease (1) or does not have it (0).

Key Characteristics:

- The target variable consists of exactly two distinct classes.
- Common algorithms used for binary classification: **Logistic Regression**, **Support Vector Machines (SVM)**, **Decision Trees**, **Random Forest**, **XGBoost**, and **Neural Networks**.

---

2. Multi-Class Classification

Definition: Multi-class classification refers to problems where the target variable consists of more than two classes, but each instance belongs to only one class. The model is required to predict one class out of multiple possible classes.

Example:

- **Handwritten Digit Recognition**: A model classifies an image of a handwritten digit into one of 10 classes (0 to 9).
- **Animal Classification**: A model predicts whether an image shows a dog, cat, bird, etc., each belonging to a distinct class.

Key Characteristics:

- The target variable consists of more than two classes (but still a single class for each instance).
- The model is trained to distinguish between each class based on the input features.

- Common algorithms used for multi-class classification: **Logistic Regression (with softmax)**, **Random Forest**, **SVM (with one-vs-rest or one-vs-one strategy)**, **XGBoost**, and **Neural Networks**.

One-vs-Rest (OvR) vs. One-vs-One (OvO) in Multi-Class Classification:

- **One-vs-Rest (OvR)**: The classifier learns multiple binary classifiers, each one distinguishing one class from all other classes.
- **One-vs-One (OvO)**: A binary classifier is trained for every pair of classes.

---

3. Multi-Label Classification

Definition: In multi-label classification, each instance can belong to more than one class at the same time. Unlike multi-class classification, where each instance belongs to only one class, multi-label classification involves predicting multiple labels (categories) for a given instance.

Example:

- **Tagging Articles**: An article can be tagged with multiple labels like "technology," "health," and "education."
- **Movie Genre Prediction**: A movie can belong to multiple genres, such as "comedy," "action," and "romance."

Key Characteristics:

- Each instance can belong to multiple classes simultaneously.
- The target variable is typically represented as a vector of binary labels (e.g., `[1, 0, 1, 0]` for a movie tagged with the first and third genres).
- Common algorithms for multi-label classification: **Binary Relevance**, **Classifier Chains**, **Label Powerset**, and adaptations of models like **Logistic Regression**, **SVM**, **Random Forest**, and **Neural Networks**.

Explanation of the Metrics:

1. **Accuracy**: The proportion of correctly classified instances to the total instances.
   - Formula: (\frac{TP + TN}{TP + TN + FP + FN})
2. **Precision**: The proportion of positive predictions that are actually correct.
   - Formula: (\frac{TP}{TP + FP})
3. **Recall (Sensitivity)**: The proportion of actual positives that are correctly identified.
   - Formula: (\frac{TP}{TP + FN})
4. **F1 Score**: The harmonic mean of precision and recall, giving a balance between them.
   - Formula: (2 \times \frac{Precision \times Recall}{Precision + Recall})
5. **AUROC (Area Under Receiver Operating Characteristic Curve)**: Measures the model's ability to distinguish between the positive and negative classes. It plots the true positive rate against the false positive rate at different thresholds.
   - A value closer to 1 indicates a good model, while 0.5 indicates a random classifier.
6. **AUPRC (Area Under Precision-Recall Curve)**: Measures the trade-off between precision and recall. It's particularly useful for imbalanced datasets.
   - A value closer to 1 indicates a good model, while 0 indicates poor performance.

Let's explore classification models:

1. **Logistic Regression**: A linear model used for binary classification, estimating the probability of a class label using a logistic function. It is efficient and interpretable for simple problems.

2. **SGD Classifier**: A linear classifier optimized with stochastic gradient descent (SGD), suitable for large-scale datasets. It can handle both regression and classification tasks.

3. **SVM Classifier**: A Support Vector Machine (SVM) finds the hyperplane that best separates classes in high-dimensional space. It is effective in high-dimensional and complex decision boundaries.

4. **KNN Classifier (k=3)**: A non-parametric model that classifies based on the majority vote of the nearest 3 neighbors. It is simple and effective but can be computationally expensive for large datasets.

5. **KNN Classifier (k=5)**: Similar to the previous model, but with 5 nearest neighbors for classification. It can provide more stable predictions than using a small k value.

6. **Gaussian Process**: A non-parametric, probabilistic model that predicts the distribution of possible outcomes using Gaussian processes. It is powerful for regression tasks with uncertainty estimation.

7. **Naive Bayes**: A probabilistic classifier based on Bayes' theorem, assuming independence between features. It is efficient and works well with categorical data, especially in text classification.

8. **Decision Tree**: A non-linear model that recursively splits the data based on feature values, forming a tree structure. It is interpretable but can overfit if not regularized.

9. **Random Forest**: An ensemble of decision trees trained on random subsets of the data, improving accuracy and robustness. It reduces overfitting by averaging multiple trees' outputs.

10. **Gradient Boosting**: An ensemble method that builds trees sequentially, each correcting the errors of the previous one. It is powerful for structured data and prone to overfitting if not tuned properly.

11. **LightGBM**: A gradient boosting framework optimized for speed and memory efficiency. It uses histogram-based learning for faster training on large datasets.

12. **XGBoost**: An efficient, scalable gradient boosting library that improves performance by regularizing the model and using boosting techniques. It is widely used in machine learning competitions.

13. **AdaBoost**: An ensemble technique that adjusts weights for misclassified instances and combines weak learners (usually decision trees). It improves performance by focusing on hard-to-classify samples.

14. **CatBoost**: A gradient boosting library that handles categorical features automatically without the need for one-hot encoding. It is fast and effective, particularly with categorical data.

15. **Bagging**: An ensemble technique that trains multiple models (e.g., decision trees) on random subsets of the data and combines their predictions. It reduces variance and improves accuracy.

16. **Voting Classifier**: Combines multiple models and makes predictions based on the majority vote (for classification) or average (for regression) of the individual models. It enhances predictive power.

17. **Stacking Classifier**: Combines predictions from multiple models (base learners) using a meta-model to make the final prediction. It aims to improve model performance by leveraging diverse models.

Each of these classifiers has its strengths and weaknesses, and choosing the right one depends on the dataset and the problem at hand.

Lets Start : Binary Classification

In [71]:
```
# Import necessary libraries
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, AdaBoostClassifier, VotingClassi
from sklearn.model_selection import train_test_split
import xgboost as xgb
import lightgbm as lgb
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, average_preci
```

In [72]:
```
# Initialize the model
model = LogisticRegression()
```

In [73]:
```
# Train the model
model.fit(X_train, y_train)
```

Out[73]:
```
LogisticRegression()
```

In [74]:
```
# Predict the labels
y_pred = model.predict(X_test)
```

In [75]:
```
y_pred
```

Out[75]:
```
array([0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0,
       0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0,
       0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0,
       1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0,
       1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0,
       1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0,
       1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1,
       0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0,
       0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1,
       1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1,
       0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1,
       0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0,
       1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1,
       0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1,
       0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1,
       1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1,
       0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0,
       0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0])
```

In [76]:
```
# Predict probabilities for AUROC and AUPRC
y_prob = model.predict_proba(X_test)[:, 1]
```

In [77]:
```
# Calculate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auroc = roc_auc_score(y_test, y_prob)
auprc = average_precision_score(y_test, y_prob)

# Print results
print('Logistic Regression:')
print(f'Accuracy: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')
print(f'AUROC: {auroc:.4f}')
print(f'AUPRC: {auprc:.4f}')
```

```
Logistic Regression:
Accuracy: 0.7609
Precision: 0.7362
Recall: 0.8047
F1 Score: 0.7689
AUROC: 0.8429
AUPRC: 0.8251
```

Let's try everything

In [78]:
```
# Define and train classifiers
models = {
    'Logistic Regression': LogisticRegression(),
    'SGD Classifier': SGDClassifier(),
    'SVM Classifier': SVC(probability=True),
    'KNN Classifier (k=3)': KNeighborsClassifier(n_neighbors=3),
    'KNN Classifier (k=5)': KNeighborsClassifier(n_neighbors=5),
    'Gaussian Process': GaussianProcessClassifier(),
    'Naive Bayes': GaussianNB(),
    'Decision Tree': DecisionTreeClassifier(),
    'Random Forest': RandomForestClassifier(),
    'Gradient Boosting': GradientBoostingClassifier(),
    'LightGBM': lgb.LGBMClassifier(),
    'XGBoost': xgb.XGBClassifier(),
    'AdaBoost': AdaBoostClassifier(),
    'Bagging': BaggingClassifier(),
    'Voting Classifier': VotingClassifier(estimators=[('rf', RandomForestClassifier()), ('svc', SVC())], voting='
    'Stacking Classifier': StackingClassifier(estimators=[('rf', RandomForestClassifier()), ('svc', SVC())], fina
}
```

In [79]:
```
models.items()
```

Out[79]:
```
dict_items([('Logistic Regression', LogisticRegression()), ('SGD Classifier', SGDClassifier()), ('SVM Classifier', SVC(probability=True)), ('KNN Classifier (k=3)', KNeighborsClassifier(n_neighbors=3)), ('KNN Classifier (k=5)', KNeighborsClassifier()), ('Gaussian Process', GaussianProcessClassifier()), ('Naive Bayes', GaussianNB()), ('Decision Tree', DecisionTreeClassifier()), ('Random Forest', RandomForestClassifier()), ('Gradient Boosting', GradientBoostingClassifier()), ('LightGBM', LGBMClassifier()), ('XGBoost', XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)), ('AdaBoost', AdaBoostClassifier()), ('Bagging', BaggingClassifier()), ('Voting Classifier', VotingClassifier(estimators=[('rf', RandomForestClassifier()), ('svc', SVC())])), ('Stacking Classifier', StackingClassifier(estimators=[('rf', RandomForestClassifier()),
                                ('svc', SVC())],
                    final_estimator=LogisticRegression()))])
```

In [80]:
```python
# Initialize dictionary to store results
results = {}

# Train and evaluate each model
for name, model in models.items():
    model.fit(X_train, y_train)

    # Predict probabilities for AUROC and AUPRC
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1] if hasattr(model, 'predict_proba') else y_pred

    # Calculate metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    auroc = roc_auc_score(y_test, y_prob)
    auprc = average_precision_score(y_test, y_prob)

    # Store results
    results[name] = {
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,
        'AUROC': auroc,
        'AUPRC': auprc
    }

    # Print results for each model
    print(f'\n{name}:')
    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print(f'AUROC: {auroc:.4f}')
    print(f'AUPRC: {auprc:.4f}')
```

```
Logistic Regression:
Accuracy: 0.7609
Precision: 0.7362
Recall: 0.8047
F1 Score: 0.7689
AUROC: 0.8429
AUPRC: 0.8251

SGD Classifier:
Accuracy: 0.6460
Precision: 0.5921
Recall: 0.9116
F1 Score: 0.7179
AUROC: 0.6490
AUPRC: 0.5835

SVM Classifier:
Accuracy: 0.5586
Precision: 0.5367
Recall: 0.7814
F1 Score: 0.6364
AUROC: 0.5835
AUPRC: 0.5841

KNN Classifier (k=3):
Accuracy: 0.6253
Precision: 0.6083
Recall: 0.6791
F1 Score: 0.6418
AUROC: 0.6553
AUPRC: 0.5996

KNN Classifier (k=5):
Accuracy: 0.6230
Precision: 0.6049
Recall: 0.6837
F1 Score: 0.6419
AUROC: 0.6642
AUPRC: 0.6210

Gaussian Process:
Accuracy: 0.6253
Precision: 0.6102
Recall: 0.6698
F1 Score: 0.6386
AUROC: 0.6589
AUPRC: 0.6480

Naive Bayes:
Accuracy: 0.7494
Precision: 0.7650
Recall: 0.7116
F1 Score: 0.7373
AUROC: 0.8386
AUPRC: 0.8274

Decision Tree:
Accuracy: 0.7333
Precision: 0.7240
Recall: 0.7442
F1 Score: 0.7339
AUROC: 0.7335
AUPRC: 0.6652

Random Forest:
Accuracy: 0.7954
Precision: 0.8058
Recall: 0.7721
F1 Score: 0.7886
AUROC: 0.8664
AUPRC: 0.8810

Gradient Boosting:
Accuracy: 0.7977
Precision: 0.7679
Recall: 0.8465
F1 Score: 0.8053
AUROC: 0.8904
AUPRC: 0.9019
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Number of positive: 663, number of negative: 641
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.000815 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1335
[LightGBM] [Info] Number of data points in the train set: 1304, number of used features: 14
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.508436 -> initscore=0.033746
[LightGBM] [Info] Start training from score 0.033746

LightGBM:
Accuracy: 0.7885
Precision: 0.7915
Recall: 0.7767
F1 Score: 0.7840
AUROC: 0.8840
AUPRC: 0.8947

XGBoost:
Accuracy: 0.7862
Precision: 0.7773
Recall: 0.7953
F1 Score: 0.7862
AUROC: 0.8816
AUPRC: 0.8964

AdaBoost:
Accuracy: 0.7954
Precision: 0.7890
Recall: 0.8000
F1 Score: 0.7945
AUROC: 0.8656
AUPRC: 0.8615

Bagging:
Accuracy: 0.7701
Precision: 0.7889
Recall: 0.7302
F1 Score: 0.7585
AUROC: 0.8337
AUPRC: 0.8347

Voting Classifier:
Accuracy: 0.7471
Precision: 0.8302
Recall: 0.6140
F1 Score: 0.7059
AUROC: 0.7456
AUPRC: 0.7005

Stacking Classifier:
Accuracy: 0.7977
Precision: 0.8009
Recall: 0.7860
F1 Score: 0.7934
AUROC: 0.8767
AUPRC: 0.8906
```

In [81]:

```python
# Create a DataFrame from the results dictionary
results_df = pd.DataFrame(results).T

results_df
```

Out[81]:

	Accuracy	Precision	Recall	F1 Score	AUROC	AUPRC
Logistic Regression	0.760920	0.736170	0.804651	0.768889	0.842875	0.825088
SGD Classifier	0.645977	0.592145	0.911628	0.717949	0.648996	0.583494
SVM Classifier	0.558621	0.536741	0.781395	0.636364	0.583488	0.584057
KNN Classifier (k=3)	0.625287	0.608333	0.679070	0.641758	0.655317	0.599589
KNN Classifier (k=5)	0.622989	0.604938	0.683721	0.641921	0.664175	0.621039
Gaussian Process	0.625287	0.610169	0.669767	0.638581	0.658922	0.648048
Naive Bayes	0.749425	0.765000	0.711628	0.737349	0.838584	0.827387
Decision Tree	0.733333	0.723982	0.744186	0.733945	0.733457	0.665214
Random Forest	0.795402	0.805825	0.772093	0.788599	0.866448	0.880998
Gradient Boosting	0.797701	0.767932	0.846512	0.805310	0.890444	0.901906
LightGBM	0.788506	0.791469	0.776744	0.784038	0.883964	0.894658
XGBoost	0.786207	0.777273	0.795349	0.786207	0.881554	0.896410
AdaBoost	0.795402	0.788991	0.800000	0.794457	0.865592	0.861471
Bagging	0.770115	0.788945	0.730233	0.758454	0.833668	0.834678
Voting Classifier	0.747126	0.830189	0.613953	0.705882	0.745613	0.700502
Stacking Classifier	0.797701	0.800948	0.786047	0.793427	0.876744	0.890621

In [82]:
```
# Sort the DataFrame by 'Accuracy' in descending order
results_df.sort_values(by='Accuracy', ascending=False)
```

Out[82]:

	Accuracy	Precision	Recall	F1 Score	AUROC	AUPRC
Gradient Boosting	0.797701	0.767932	0.846512	0.805310	0.890444	0.901906
Stacking Classifier	0.797701	0.800948	0.786047	0.793427	0.876744	0.890621
Random Forest	0.795402	0.805825	0.772093	0.788599	0.866448	0.880998
AdaBoost	0.795402	0.788991	0.800000	0.794457	0.865592	0.861471
LightGBM	0.788506	0.791469	0.776744	0.784038	0.883964	0.894658
XGBoost	0.786207	0.777273	0.795349	0.786207	0.881554	0.896410
Bagging	0.770115	0.788945	0.730233	0.758454	0.833668	0.834678
Logistic Regression	0.760920	0.736170	0.804651	0.768889	0.842875	0.825088
Naive Bayes	0.749425	0.765000	0.711628	0.737349	0.838584	0.827387
Voting Classifier	0.747126	0.830189	0.613953	0.705882	0.745613	0.700502
Decision Tree	0.733333	0.723982	0.744186	0.733945	0.733457	0.665214
SGD Classifier	0.645977	0.592145	0.911628	0.717949	0.648996	0.583494
KNN Classifier (k=3)	0.625287	0.608333	0.679070	0.641758	0.655317	0.599589
Gaussian Process	0.625287	0.610169	0.669767	0.638581	0.658922	0.648048
KNN Classifier (k=5)	0.622989	0.604938	0.683721	0.641921	0.664175	0.621039
SVM Classifier	0.558621	0.536741	0.781395	0.636364	0.583488	0.584057

Let's try : Multiclass Classification

In [83]:
```
m_cls_df.head()
```

Out[83]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [85]:
```
# Split the data into features (X) and target (y)
X = m_cls_df.drop('species', axis=1)
y = m_cls_df['species']

# Encode the target labels
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
```

In [86]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

# Initialize the Gradient Boosting model
model = LogisticRegression()

# Train the model
model.fit(X_train, y_train)

# Predict the values
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]

# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.4f}')
```

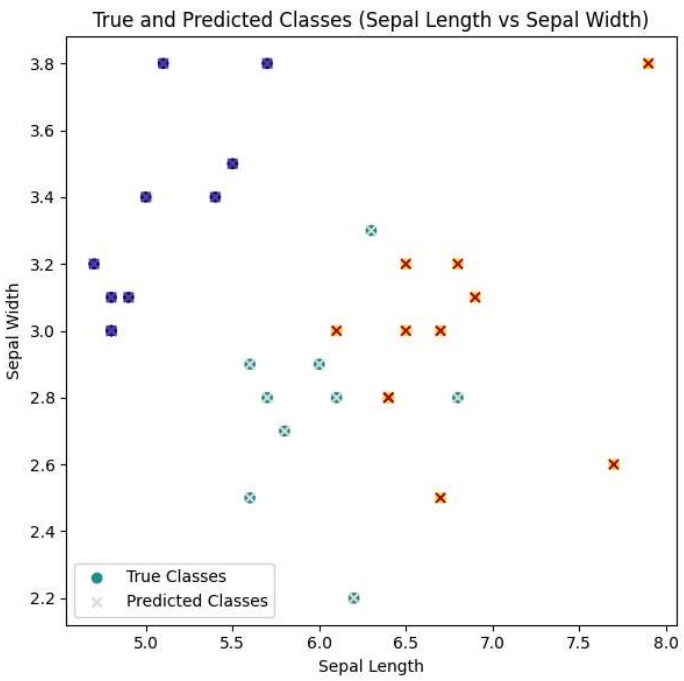
```
Accuracy: 1.0000
```

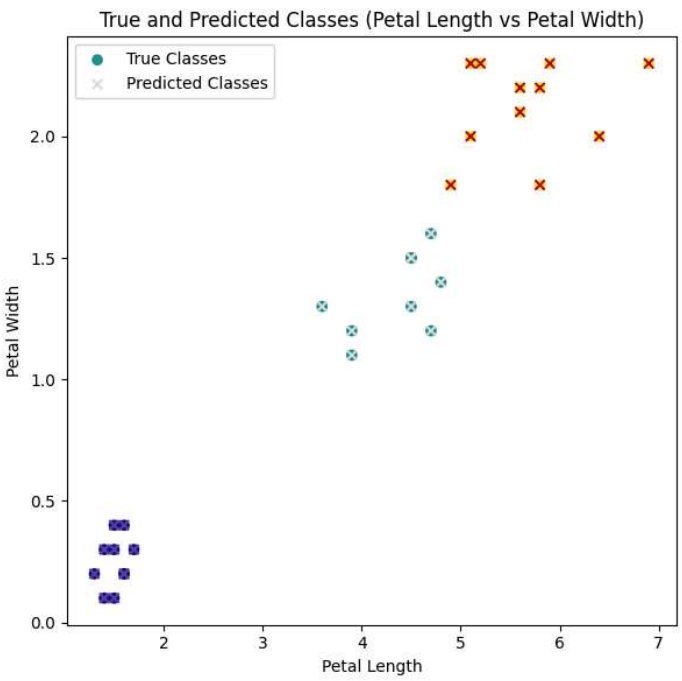
In [87]:
```
# Plot the data and predictions
plt.figure(figsize=(12, 6))

# Plot (x = sepal_length, y = sepal_width) to show true values
plt.subplot(1, 2, 1)
plt.scatter(X_test['sepal_length'], X_test['sepal_width'], c=y_test, cmap='viridis', label='True Classes')
plt.scatter(X_test['sepal_length'], X_test['sepal_width'], c=y_pred, cmap='coolwarm', marker='x', label='Predicte
plt.title('True and Predicted Classes (Sepal Length vs Sepal Width)')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend()

# Plot (x = petal_length, y = petal_width) to show true values
plt.subplot(1, 2, 2)
plt.scatter(X_test['petal_length'], X_test['petal_width'], c=y_test, cmap='viridis', label='True Classes')
plt.scatter(X_test['petal_length'], X_test['petal_width'], c=y_pred, cmap='coolwarm', marker='x', label='Predicte
plt.title('True and Predicted Classes (Petal Length vs Petal Width)')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.legend()

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



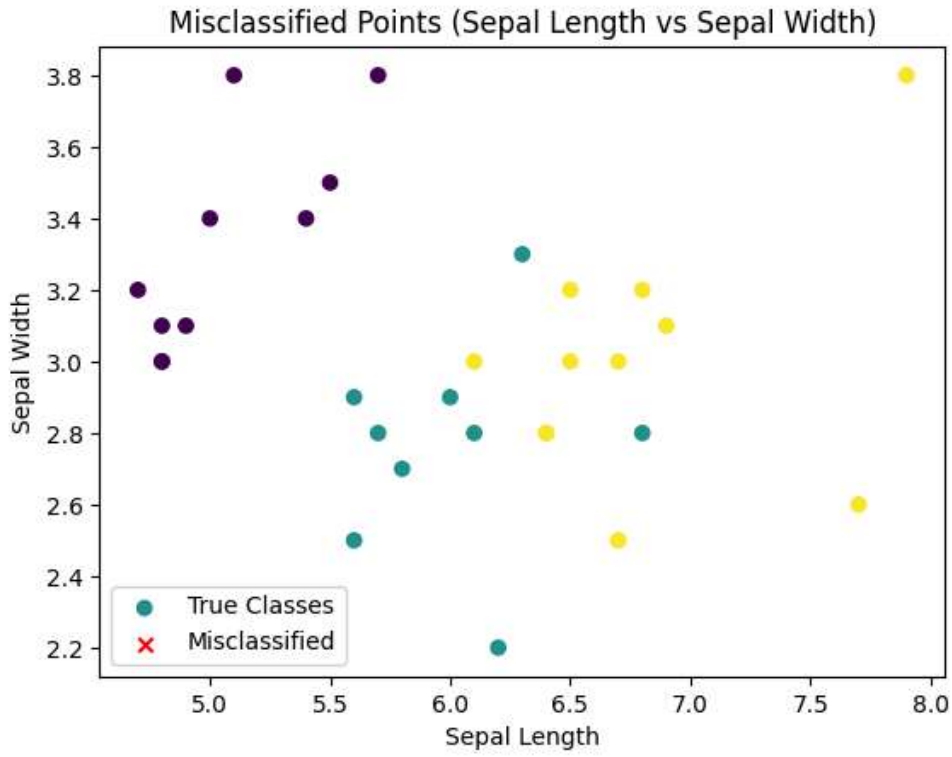


In [88]:
```
# Confusion Matrix to see the misclassifications
conf_matrix = confusion_matrix(y_test, y_pred)
print(f'Confusion Matrix:\n{conf_matrix}')

# Highlight misclassified points
misclassified_idx = np.where(y_test != y_pred)[0]
misclassified_points = X_test.iloc[misclassified_idx]

# Plot misclassified points
plt.scatter(X_test['sepal_length'], X_test['sepal_width'], c=y_test, cmap='viridis', label='True Classes')
plt.scatter(misclassified_points['sepal_length'], misclassified_points['sepal_width'], color='red', marker='x', l
plt.title('Misclassified Points (Sepal Length vs Sepal Width)')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend()
plt.show()
```

```
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```



7. Regression Model

Basic Models

Here are brief definitions of each model and evaluation metric:

- **Linear Regression**: A simple linear approach to modeling the relationship between one or more features and a continuous target. It minimizes the sum of squared errors between the observed and predicted values.
- **Ridge Regression**: A linear regression model with an L2 regularization term that penalizes large coefficients to prevent overfitting, improving generalization.
- **Lasso Regression**: A variant of linear regression that uses L1 regularization to encourage sparsity in the coefficients, potentially leading to feature selection.
- **ElasticNet**: A combination of L1 and L2 regularization, balancing between ridge and lasso regression, and works well when there are correlations between features.
- **LARS Lasso**: A variant of the Lasso model that uses the Least Angle Regression (LARS) algorithm for efficient computation when performing L1 regularization.
- **Bayesian Ridge**: A regression model that assumes a probabilistic approach, treating the coefficients as random variables and using a Bayesian framework to estimate them.
- **SGD Regressor (Stochastic Gradient Descent)**: A regression model that uses stochastic optimization to minimize the error between predicted and actual values, often faster than traditional methods.
- **Kernel Ridge Regression**: A regression model that combines ridge regression with the kernel trick to handle non-linear relationships by transforming input space into a higher-dimensional space.
- **Support Vector Machine (SVR)**: A regression model that finds the hyperplane that best fits the data, using a kernel function to model complex relationships.
- **Polynomial Regression**: A linear regression model extended to polynomial features, allowing the model to fit non-linear relationships between the features and the target.
- **Gaussian Naive Bayes**: A probabilistic model assuming that the features follow a Gaussian (normal) distribution, often used for classification but can be adapted for regression.
- **Decision Tree Regressor**: A model that splits the data into segments using decision rules, learning the data distribution recursively to make predictions.
- **Random Forest Regressor**: An ensemble of decision trees, where multiple trees are trained and the average prediction is used to improve performance and reduce overfitting.
- **Voting Regressor**: An ensemble model that combines multiple base models (e.g., decision trees, random forests) and takes the average of their predictions for better accuracy.
- **Gradient Boosting**: An ensemble technique that builds trees sequentially, each one correcting errors made by the previous tree, optimizing for the best prediction.
- **LightGBM**: A gradient boosting framework that uses histogram-based algorithms for faster training, optimized for large datasets with high-dimensional features.
- **XGBoost**: A high-performance gradient boosting algorithm that uses regularization to prevent overfitting, making it one of the most popular models in machine learning competitions.
- **AdaBoost Regressor**: An ensemble model that combines multiple weak learners (typically decision trees), focusing on the errors of previous models to improve performance.

Metrics

1. **MAE (Mean Absolute Error)**: Measures the average of the absolute differences between predicted and actual values. Lower MAE indicates better model performance.

2. **MAPE (Mean Absolute Percentage Error)**: Calculates the average of absolute percentage errors between predictions and actual values. It is useful when comparing models across different scales.

3. **MSE (Mean Squared Error)**: Measures the average of squared differences between predicted and actual values. MSE penalizes larger errors more than MAE.

4. **RMSE (Root Mean Squared Error)**: The square root of MSE, providing a more interpretable result in the same units as the target variable.

5. **R2 Score (R-squared)**: Measures the proportion of variance in the target variable that is explained by the model. A higher R2 score indicates a better fit.

Data Processing

In [91]:
```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

In [89]:
```
reg_df.head()
```

Out[89]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	I
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	

5 rows × 81 columns

In [90]:
```
# 1. Separate features and target
X = reg_df.drop(columns='SalePrice')
y = reg_df['SalePrice']
```

In [92]:
```
# 2. Identify categorical and numerical columns
categorical_columns = X.select_dtypes(include=['object']).columns
numerical_columns = X.select_dtypes(exclude=['object']).columns
```

In [93]:
```
# 3. Handle missing values
# - For categorical variables: Impute with the most frequent value
# - For numerical variables: Impute with the median value
numerical_imputer = SimpleImputer(strategy='median')
categorical_imputer = SimpleImputer(strategy='most_frequent')

# 4. Encoding categorical variables
# - Label Encoding for binary categories
# - OneHotEncoding for multi-category columns
categorical_transformer = Pipeline(steps=[
    ('imputer', categorical_imputer),
    ('encoder', OneHotEncoder(handle_unknown='ignore'))
])
```

In [94]:
```
# 5. Scaling numerical features
numerical_transformer = Pipeline(steps=[
    ('imputer', numerical_imputer),
    ('scaler', StandardScaler())
])
```

In [95]:
```
# 6. Combine transformers using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_columns),
        ('cat', categorical_transformer, categorical_columns)
    ]
)
```

In [96]:
```
# Apply preprocessing to the features
X_processed = preprocessor.fit_transform(X)
```

In [97]:
```
# Now, X_processed is ready to be used for model training
# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_processed, y, test_size=0.2, random_state=42)
```

Modleing : Linear Reg

In [98]:
```
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet, BayesianRidge, LassoLars, SGDRegress
from sklearn.kernel_ridge import KernelRidge
from sklearn.svm import SVR
from sklearn.preprocessing import PolynomialFeatures
from sklearn.naive_bayes import GaussianNB, MultinomialNB, ComplementNB, BernoulliNB
```

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, VotingRegressor, GradientBoostingRegressor, AdaBoostRegressor
from sklearn.metrics import mean_absolute_error, mean_absolute_percentage_error, mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import lightgbm as lgb
import xgboost as xgb
```

In [99]:
```
# Initialize the Linear Regression model
linear_reg_model = LinearRegression()

# Fit the model on the training data
linear_reg_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = linear_reg_model.predict(X_test)
```

In [100…
```
# Calculate performance metrics
mse = mean_squared_error(y_test, y_pred)
rmse = mse ** 0.5
r2 = r2_score(y_test, y_pred)

# Print the performance metrics
print(f'Mean Squared Error (MSE): {mse:.4f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')
print(f'R-squared (R2): {r2:.4f}')
```

```
Mean Squared Error (MSE): 873643524.4822
Root Mean Squared Error (RMSE): 29557.4614
R-squared (R2): 0.8861
```

Let's try everything

In [101…
```
# Define all models
models = {
    'Linear Regression': LinearRegression(),
    'Ridge Regression': Ridge(),
    'Lasso Regression': Lasso(),
    'ElasticNet': ElasticNet(),
    'LARS Lasso': LassoLars(),
    'Bayesian Ridge': BayesianRidge(),
    'SGD Regressor': make_pipeline(StandardScaler(), SGDRegressor()),
    'Kernel Ridge Regression': KernelRidge(),
    'Support Vector Machines (SVR)': make_pipeline(StandardScaler(), SVR()),
    'Polynomial Regression': make_pipeline(PolynomialFeatures(degree=2), LinearRegression()),
    'Gaussian Naive Bayes': GaussianNB(),
    'Decision Tree Regressor': DecisionTreeRegressor(),
    'Random Forest Regressor': RandomForestRegressor(),
    'Voting Regressor': VotingRegressor(estimators=[('rf', RandomForestRegressor()), ('dt', DecisionTreeRegressor
    'Gradient Boosting': GradientBoostingRegressor(),
    'LightGBM': lgb.LGBMRegressor(),
    'XGBoost': xgb.XGBRegressor(),
    'AdaBoost Regressor': AdaBoostRegressor()
}
```

In [102…
```
# Initialize dictionary to store results
results = {}

# Train and evaluate each model
for name, model in models.items():
    # Fit the model on the training data
    model.fit(X_train.toarray(), y_train)

    # Predict on the test data
    y_pred = model.predict(X_test.toarray())

    # Calculate metrics
    mae = mean_absolute_error(y_test, y_pred)
    mape = mean_absolute_percentage_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, y_pred)

    # Store the results
    results[name] = {
        'MAE': mae,
        'MAPE': mape,
```

```
        'MSE': mse,
        'RMSE': rmse,
        'R2 Score': r2
    }
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.001687 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 3457
[LightGBM] [Info] Number of data points in the train set: 1168, number of used features: 179
[LightGBM] [Info] Start training from score 181441.541952
```

In [103…

```
# Convert results into a DataFrame
results_df = pd.DataFrame(results).T
results_df
```

Out[103]:

	MAE	MAPE	MSE	RMSE	R2 Score
Linear Regression	18262.222435	0.112561	8.737078e+08	2.955855e+04	0.886092
Ridge Regression	19112.700563	0.117481	8.795545e+08	2.965728e+04	0.885330
Lasso Regression	17863.329691	0.110385	7.940583e+08	2.817904e+04	0.896477
ElasticNet	20179.430874	0.114003	1.320102e+09	3.633321e+04	0.827895
LARS Lasso	17075.720142	0.105560	7.824198e+08	2.797177e+04	0.897994
Bayesian Ridge	19051.051605	0.116686	9.424256e+08	3.069895e+04	0.877134
SGD Regressor	835996.907221	6.222005	1.441437e+12	1.200598e+06	-186.923765
Kernel Ridge Regression	19091.411199	0.117222	8.750817e+08	2.958178e+04	0.885913
Support Vector Machines (SVR)	59543.746408	0.359564	7.858734e+09	8.864950e+04	-0.024563
Polynomial Regression	21541.004816	0.136332	1.018965e+09	3.192123e+04	0.867155
Gaussian Naive Bayes	43532.400685	0.245303	5.307560e+09	7.285300e+04	0.308040
Decision Tree Regressor	28456.863014	0.166177	1.926827e+09	4.389564e+04	0.748795
Random Forest Regressor	17214.238870	0.104355	8.694746e+08	2.948685e+04	0.886644
Voting Regressor	21154.282106	0.125119	1.056179e+09	3.249891e+04	0.862303
Gradient Boosting	16544.600170	0.098612	6.964074e+08	2.638953e+04	0.909208
LightGBM	17364.951784	0.106163	8.831810e+08	2.971836e+04	0.884857
XGBoost	17493.976562	0.104716	7.615373e+08	2.759596e+04	0.900716
AdaBoost Regressor	25001.365645	0.173339	1.257150e+09	3.545631e+04	0.836102

In [104…

```
# Sort the results by R2 Score in descending order
results_df.sort_values(by='R2 Score', ascending=False)
```

Out[104]:

	MAE	MAPE	MSE	RMSE	R2 Score
Gradient Boosting	16544.600170	0.098612	6.964074e+08	2.638953e+04	0.909208
XGBoost	17493.976562	0.104716	7.615373e+08	2.759596e+04	0.900716
LARS Lasso	17075.720142	0.105560	7.824198e+08	2.797177e+04	0.897994
Lasso Regression	17863.329691	0.110385	7.940583e+08	2.817904e+04	0.896477
Random Forest Regressor	17214.238870	0.104355	8.694746e+08	2.948685e+04	0.886644
Linear Regression	18262.222435	0.112561	8.737078e+08	2.955855e+04	0.886092
Kernel Ridge Regression	19091.411199	0.117222	8.750817e+08	2.958178e+04	0.885913
Ridge Regression	19112.700563	0.117481	8.795545e+08	2.965728e+04	0.885330
LightGBM	17364.951784	0.106163	8.831810e+08	2.971836e+04	0.884857
Bayesian Ridge	19051.051605	0.116686	9.424256e+08	3.069895e+04	0.877134
Polynomial Regression	21541.004816	0.136332	1.018965e+09	3.192123e+04	0.867155
Voting Regressor	21154.282106	0.125119	1.056179e+09	3.249891e+04	0.862303
AdaBoost Regressor	25001.365645	0.173339	1.257150e+09	3.545631e+04	0.836102
ElasticNet	20179.430874	0.114003	1.320102e+09	3.633321e+04	0.827895
Decision Tree Regressor	28456.863014	0.166177	1.926827e+09	4.389564e+04	0.748795
Gaussian Naive Bayes	43532.400685	0.245303	5.307560e+09	7.285300e+04	0.308040
Support Vector Machines (SVR)	59543.746408	0.359564	7.858734e+09	8.864950e+04	-0.024563
SGD Regressor	835996.907221	6.222005	1.441437e+12	1.200598e+06	-186.923765

8. Time Series

Check these notebooks:

- 🗓️ Time Series Analysis 🔺 Forecasting 🔻
- Intro to LSTM 🗓️ Time Series Forecasting

Conclusion

Thanks

If you use it, cite:

Azmine Toushik Wasi. (2024). CIOL Presnts Winer ML BootCamp. https://github.com/ciol-researchlab/CIOL-Winter-ML-Bootcamp

```
@misc{wasi2024CIOL-WMLB,
      title={CIOL Presnts Winer ML BootCamp},
      author={Azmine Toushik Wasi},
      year={2024},
      url={https://github.com/ciol-researchlab/CIOL-Winter-ML-Bootcamp},
}
```