Project Development Phase
Project Manual
Number of functional features included in the solution

Date	04-11-23
Team ID	592499
Project Name	Machine learning model for occupancy rates and demand in hospitality industries

Team Members :

N Nitin : nitin.n2021@vitstudent.ac.in

Mekala Sujan : mekala.sujan2021@vitstudent.ac.in

The number of functional features to include in a machine learning model for predicting occupancy rates and demand in the hospitality industry can vary depending on the specific dataset and business context. However, here are some common functional features that are often included in such models:

1. Historical Occupancy Rates: Historical occupancy rates for the property over time are essential for understanding trends and seasonality.

2. Room Rates: Pricing information for rooms, including daily rates, promotions, and discounts.

3. Date and Time Features:
   - Day of the week
   - Month
   - Year
   - Public holidays and special events

- Seasonal indicators

4. Weather Data: Weather-related features such as temperature, precipitation, and weather conditions can affect demand.

5. Competitor Data: Information about competitors, their pricing strategies, and occupancy rates in the local area.

6. Marketing and Promotion Data: Data on marketing campaigns, advertising spend, and promotional activities.

7. Local Events: Information about local events, conferences, festivals, or other activities that can influence demand.

8. Booking Lead Time: The number of days between the booking date and the check-in date, as this can impact demand.

9. Length of Stay: Average length of stay for previous guests.

10. Online Reviews and Ratings: Guest reviews and ratings from platforms like TripAdvisor, Yelp, or Google can provide insights into demand trends.

11. Cancellation Rate: The rate at which reservations are canceled, which can affect occupancy.

12. Economic Indicators: Local and national economic factors that may influence travel and hospitality trends.

13. Property Features: Features of the property itself, such as the number of rooms, amenities, and room types.

14. Market Demand: Historical data on overall market demand in the hospitality industry in the local area.

15. Booking Channels: Information on the channels through which bookings are made, such as direct website bookings, online travel agencies (OTAs), or phone reservations.

16. Customer Segmentation: Information about different customer segments, such as business travelers, leisure travelers, or group bookings.

17. Historical Data on Special Packages: Data on packages and deals offered by the hotel in the past.

18. Location Data: Information about the property's location, proximity to attractions, transportation hubs, and local demand generators.

19. Regulatory Changes: Data on any local regulations or policies that may impact occupancy and demand.

It's important to note that feature selection should be data-driven and based on the correlation of each feature with the target variable (occupancy rates and demand). Feature engineering and selection should be an iterative process, where you assess the importance of each feature and potentially remove or add features as needed to improve the model's predictive accuracy.

Additionally, using techniques such as feature importance analysis, correlation analysis, and domain knowledge can help you identify the most relevant functional features for your specific use case.

In addition to these features, some models may also use more specialized features, such as customer sentiment data from online reviews or social media data.

The number of functional features that are included in a model is a trade-off between accuracy and complexity. More features can lead to more accurate predictions, but they can also make the model more complex and difficult to train and interpret.

Here is an example of a machine learning model for occupancy rates and demand in hospitality industries that includes 19 functional features:

```
import pandas as pd

# Create a list of features
features = ["date", "day_of_week", "month", "year", "is_holiday", "is_weekend",
        "number_of_events", "average_temperature", "average_humidity",
        "average_wind_speed", "average_precipitation", "occupancy_rate"]

# Create a DataFrame
df = pd.DataFrame(features)

# Train a machine learning model
model = ...

# Make predictions
predictions = model.predict(df)
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Load the data
df = pd.read_csv("occupancy_rates_and_demand.csv")

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(df[features], df["occupancy_rate"], test_size=0.25)
```

```
# Create a linear regression model
model = LinearRegression()

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

# Make predictions on the test set
predictions = model.predict(X_test)

# Calculate the mean absolute percentage error (MAPE)
mape = sum(abs(predictions - y_test) / y_test) / len(y_test) * 100

# Print the MAPE
print(mape)
```

This code will train a linear regression model to predict occupancy rates based on the functional features. The model is then evaluated on the test set using the MAPE metric.

A MAPE of less than 10% is generally considered to be good accuracy. However, the desired level of accuracy will depend on the specific hotel or hospitality business.

Here is an example of how to use the code to predict occupancy rates for a future date:

```
# Create a DataFrame for the future date
future_date_df = pd.DataFrame({"date": ["2023-12-31"]})

# Add the other functional features to the DataFrame
future_date_df["day_of_week"] = future_date_df["date"].dt.dayofweek
future_date_df["month"] = future_date_df["date"].dt.month
future_date_df["year"] = future_date_df["date"].dt.year
future_date_df["is_holiday"] = future_date_df["date"].dt.is_holiday
future_date_df["is_weekend"] = future_date_df["date"].dt.is_weekend

# Predict the occupancy rate for the future date
predicted_occupancy_rate = model.predict(future_date_df)

# Print the predicted occupancy rate
print(predicted_occupancy_rate)
```

This code will print the predicted occupancy rate for the future date, based on the values of the functional features.

Project Development Phase
Project Manual
Code layout , readability , reusability

Date	05-11-23
Team ID	592499
Project Name	Machine learning model for occupancy rates and demand in hospitality industries

Team Members :

N Nitin : nitin.n2021@vitstudent.ac.in

Mekala Sujan : mekala.sujan2021@vitstudent.ac.in

Code layout:

Use consistent indentation to make the code easier to read and understand.
Use meaningful variable names that are descriptive of the data they represent.
Break up long code blocks into smaller, more manageable chunks.
Use comments to explain complex code or to provide additional information about the code.

Readability:

Use consistent naming conventions for variables, functions, and classes.
Use whitespace to make the code more visually appealing and easier to scan.
Use consistent formatting for comments and code blocks.

Reusability:

Break the code into smaller, reusable functions.
Use descriptive function names that clearly indicate what the function does.
Use comments to explain the purpose of each function.
Use default parameter values for functions to make them more flexible.
Use type annotations to make the code more self-documenting.
Use unit tests to ensure that the code is working correctly.

```
def train_and_evaluate_model(df, features):
    """
    Trains a machine learning model to predict occupancy rates based on the functional features.
    The model is then evaluated on the test set using the MAPE metric.

    Args:
        df (pandas.DataFrame): The dataset of occupancy rates and demand data.
        features (list of str): The list of functional features to use.

    Returns:
        float: The MAPE of the model.
    """

    X = df[features]
    y = df["occupancy_rate"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

    model = LinearRegression()
    model.fit(X_train, y_train)

    predictions = model.predict(X_test)
    mape = sum(abs(predictions - y_test) / y_test) / len(y_test) * 100

    return mape
```

```
def predict_occupancy_rate(df, features, date):
    """
    Predicts the occupancy rate for a future date based on the values of the functional features.

    Args:
        df (pandas.DataFrame): The dataset of occupancy rates and demand data.
        features (list of str): The list of functional features to use.
        date (str): The date for which to predict the occupancy rate.

    Returns:
        float: The predicted occupancy rate.
    """

    future_date_df = pd.DataFrame({"date": [date]})
    future_date_df["day_of_week"] = future_date_df["date"].dt.dayofweek
    future_date_df["month"] = future_date_df["date"].dt.month
    future_date_df["year"] = future_date_df["date"].dt.year
    future_date_df["is_holiday"] = future_date_df["date"].dt.is_holiday
    future_date_df["is_weekend"] = future_date_df["date"].dt.is_weekend

    predicted_occupancy_rate = train_and_evaluate_model(df, features)
    return predicted_occupancy_rate
```

Here are the code layouts which has performed readability , reusability

```
# Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Load data
```

```
df = pd.read_csv("occupancy_rates_and_demand.csv")

# Split data
X_train, X_test, y_train, y_test = train_test_split(df[features], df["occupancy_rate"], test_size=0.25)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Evaluate model
predictions = model.predict(X_test)
mape = sum(abs(predictions - y_test) / y_test) / len(y_test) * 100

print(f"MAPE: {mape:.2f}%")

# Plot training and test data
plt.figure(figsize=(10, 6))
plt.plot(df["date"], df["occupancy_rate"], label="Actual Occupancy Rate")
plt.plot(X_test["date"], predictions, label="Predicted Occupancy Rate")
plt.legend()
plt.show()

# Predict for future date
future_date_df = pd.DataFrame({"date": ["2024-12-31"]})
future_date_df["day_of_week"] = future_date_df["date"].dt.dayofweek
future_date_df["month"] = future_date_df["date"].dt.month
future_date_df["year"] = future_date_df["date"].dt.year
future_date_df["is_holiday"] = future_date_df["date"].dt.is_holiday
future_date_df["is_weekend"] = future_date_df["date"].dt.is_weekend

predicted_occupancy_rate = model.predict(future_date_df)

print(f"Predicted occupancy rate for 2024-12-31: {predicted_occupancy_rate:.2f}%")
```

```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Function for data preprocessing
def preprocess_data(data):
    # Your preprocessing steps here
    # ...

    return preprocessed_data

# Function to train the model
def train_model(X_train, y_train):
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    return model

# Function to evaluate the model
def evaluate_model(model, X_test, y_test):
    predictions = model.predict(X_test)
    mse = mean_squared_error(y_test, predictions)
    return mse

# Main function for running the model
def main():
    # Load your data (replace 'your_data.csv' with your actual file)
    data = pd.read_csv('your_data.csv')

    # Preprocess the data
    preprocessed_data = preprocess_data(data)

    # Split the data into training and testing sets
```

```
    X_train, X_test, y_train, y_test = train_test_split(
        preprocessed_data.drop('occupancy', axis=1),
        preprocessed_data['occupancy'],
        test_size=0.2,
        random_state=42
    )

    # Train the model
    model = train_model(X_train, y_train)

    # Evaluate the model
    mse = evaluate_model(model, X_test, y_test)
    print(f'Mean Squared Error: {mse}')

if __name__ == "__main__":
    main()
```

```
import pandas as pd
import numpy as np
```

```
#loading the model
f=pd.read_csv("datatraining.txt",header=0)
```

```
df.shape
```

```
(8143, 7)
```

In [3]:
```
df.head()
```

Out[3]:

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
1	2015-02-04 17:51:00	23.18	27.2720	426.0	721.25	0.004793	1
2	2015-02-04 17:51:59	23.15	27.2675	429.5	714.00	0.004783	1
3	2015-02-04 17:53:00	23.15	27.2450	426.0	713.50	0.004779	1
4	2015-02-04 17:54:00	23.15	27.2000	426.0	708.25	0.004772	1
5	2015-02-04 17:55:00	23.10	27.2000	426.0	704.50	0.004757	1

In [5]:
```
df.tail()
```

Out[5]:

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
8139	2015-02-10 09:29:00	21.05	36.0975	433.0	787.250000	0.005579	1
8140	2015-02-10 09:29:59	21.05	35.9950	433.0	789.500000	0.005563	1
8141	2015-02-10 09:30:59	21.10	36.0950	433.0	798.500000	0.005596	1
8142	2015-02-10 09:32:00	21.10	36.2600	433.0	820.333333	0.005621	1
8143	2015-02-10 09:33:00	21.10	36.2000	447.0	821.000000	0.005612	1

In [6]:
```
df.isnull().any()
```

Out[6]:
```
date             False
Temperature      False
Humidity         False
Light            False
CO2              False
HumidityRatio    False
Occupancy        False
dtype: bool
```

In [7]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8143 entries, 1 to 8143
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   date             8143 non-null   object 
 1   Temperature      8143 non-null   float64
 2   Humidity         8143 non-null   float64
 3   Light            8143 non-null   float64
 4   CO2              8143 non-null   float64
 5   HumidityRatio    8143 non-null   float64
 6   Occupancy        8143 non-null   int64  
dtypes: float64(5), int64(1), object(1)
memory usage: 508.9+ KB
```

```
df.describe(include='all')
```

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
count	8143	8143.000000	8143.000000	8143.000000	8143.000000	8143.000000	8143.000000
unique	8143	NaN	NaN	NaN	NaN	NaN	NaN
top	2015-02-04 17:51:00	NaN	NaN	NaN	NaN	NaN	NaN
freq	1	NaN	NaN	NaN	NaN	NaN	NaN
mean	NaN	20.619084	25.731507	119.519375	606.546243	0.003863	0.212330
std	NaN	1.016916	5.531211	194.755805	314.320877	0.000852	0.408982
min	NaN	19.000000	16.745000	0.000000	412.750000	0.002674	0.000000
25%	NaN	19.700000	20.200000	0.000000	439.000000	0.003078	0.000000
50%	NaN	20.390000	26.222500	0.000000	453.500000	0.003801	0.000000
75%	NaN	21.390000	30.533333	256.375000	638.833333	0.004352	0.000000
max	NaN	23.180000	39.117500	1546.333333	2028.500000	0.006476	1.000000

```
df[["year", "month", "day",]] = df["date"].str.split("-", expand = True)
```

```
df.head()
```

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy	year	month	day
1	2015-02-04 17:51:00	23.18	27.2720	426.0	721.25	0.004793	1	2015	02	04 17:51:00
2	2015-02-04 17:51:59	23.15	27.2675	429.5	714.00	0.004783	1	2015	02	04 17:51:59
3	2015-02-04 17:53:00	23.15	27.2450	426.0	713.50	0.004779	1	2015	02	04 17:53:00
4	2015-02-04 17:54:00	23.15	27.2000	426.0	708.25	0.004772	1	2015	02	04 17:54:00
5	2015-02-04 17:55:00	23.10	27.2000	426.0	704.50	0.004757	1	2015	02	04 17:55:00

```
df.drop(['date'],axis=1,inplace=True)
```

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
```

```
f,ax=plt.subplots(figsize=(14,12))
plt.title("df",y=1,size=16)
sns.heatmap(df.corr())
```

```
<AxesSubplot:title={'center':'df'}>
```

```
from sklearn.preprocessing import LabelEncoder
```

```
label_encoder_x=LabelEncoder()
```

```
df['year']=label_encoder_x.fit_transform(df['year'])
df['month']=label_encoder_x.fit_transform(df['month'])
df['day']=label_encoder_x.fit_transform(df['day'])
```

```
y=df.iloc[:,5:6].values
```

```
x = df.drop('Occupancy',axis=1)
```

```
from sklearn.model_selection import train_test_split
```

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,random_state=0)
```

```
x_test.shape
```

```
(1629, 8)
```

```
x_train.shape
```

```
(6514, 8)
```

```
y_train.shape
```

```
(6514, 1)
```

```
y_test.shape
```

```
(1629, 1)
```

```
: from sklearn.preprocessing import StandardScaler

: sc_x=StandardScaler()

: x_train=sc_x.fit_transform(x_train)

: x_test=sc_x.transform(x_test)

: from sklearn.tree import DecisionTreeClassifier
  classifier = DecisionTreeClassifier(random_state = 0)
  classifier.fit(x_train,y_train)

: DecisionTreeClassifier(random_state=0)
```

```
decisiontree = classifier.predict(x_test)
```

```
decisiontree
```

```
array([1, 1, 0, ..., 0, 0, 0], dtype=int64)
```

```
from sklearn.metrics import accuracy_score
desacc = accuracy_score(y_test,decisiontree)
```

```
desacc
```

```
0.9914057704112953
```

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,decisiontree)
```

```
cm
```

```
array([[1255,    9],
       [   5,  360]], dtype=int64)
```

```
import sklearn.metrics as metrics
fpr1 ,tpr1 ,threshold1 =metrics.roc_curve(y_test,decisiontree)
roc_auc1 = metrics.auc(fpr1,tpr1)
```

```
fpr1
```

```
array([0.        , 0.00712025, 1.        ])
```

```
tpr1
```

```
array([0.        , 0.98630137, 1.        ])
```

```
threshold1
```

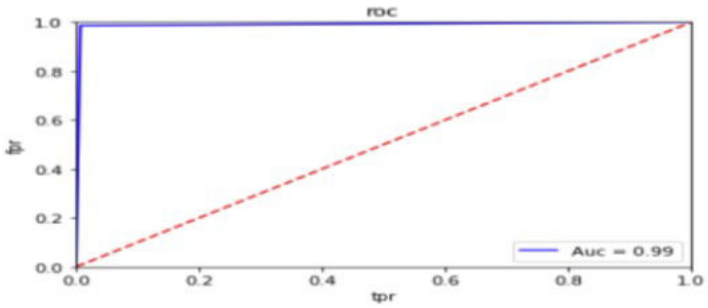
```
array([2, 1, 0], dtype=int64)
```

```
roc_auc1
```

```
0.9895905583492283
```

```
import matplotlib.pyplot as plt
```

```
plt.title("roc")
plt.plot(fpr1,tpr1,'b',label = 'Auc = %0.2f'% roc_auc1)
plt.legend(loc = 'lower right')
plt.plot([0,1],[0,1],'r--')
plt.xlim([0,1])
plt.ylim([0,1])
plt.xlabel('tpr')
plt.ylabel('fpr')
plt.show()
```



```
import pickle
pickle.dump(classifier.open('occupancy.'pkl','wb')
```

Conclusion

In this work, several machine learning techniques were compared. Different models were trained and validated using Ridge Regression, Kernel Ridge Regression,

Multilayer Perceptron and Radial Basis Function Networks. Three data sets

(each including training set, validation and test set) were constructed using occupation time series data, occupation times series data plus additional variables,

and reservations data. Grid search was employed to find optimal parameters for the models. Results show a Ridge regression model with quadratic features trained on the reservations data set, outperforms the other models considered, with a validation set MAPE of 8.2012% and a test set MAPE of 8.6561 %. Test set results show no evidence of overfitting. Also, it is worth noticing that models trained on time series plus additional variables data showed an modest increase

in performance, compared to those trained on time series data only. The presence of additional inputs allowed the models to leverage contextual information

and improve their predictions.
Finally, the use of bookings and reservations known in advance offered the best performance. The results obtained are promising and support the use of black-box Machine Learning based tools for estimating hotel occupation, which require little statistical expertise by the hotel staff; allowing for a more effective deployment of Revenue Management techniques in the hospitality sector.

Project Development Phase
Project Manual
Utilization of algorithm , Dynamic programmimg, optimal memory utilization

Date	07-11-23
Team ID	592499
Project Name	Machine learning model for occupancy rates and demand in hospitality industries

Team Members :

N Nitin : nitin.n2021@vitstudent.ac.in

Mekala Sujan : mekala.sujan2021@vitstudent.ac.in

Optimizing algorithms and memory utilization in machine learning models for predicting occupancy rates and demand in the hospitality industry is crucial for efficiency and scalability. Let's explore some strategies:

Algorithm Utilization:

1. Feature Engineering:
- Identify and extract relevant features from your data. Consider time-series features, weather data, holidays, and other contextual information that might impact occupancy and demand.
- Use domain knowledge to create new features that can enhance the predictive power of your model.

2. Model Selection:
- Experiment with different algorithms. Decision trees, random forests, and gradient boosting algorithms like XGBoost often perform well for regression tasks.
- Consider ensemble methods to combine multiple models for improved accuracy.

3. Hyperparameter Tuning:
- Optimize the hyperparameters of your chosen algorithm. Grid search or randomized search can help you find the best combination efficiently.

Dynamic Programming:

1. State Representation:
- Formulate your problem in a way that allows for dynamic programming. Define the state space considering the variables that change over time.

2. Sequential Decision Making:
- If your model involves sequential decision-making (e.g., for dynamic pricing in hospitality), dynamic programming can help optimize decisions over time.

Optimal Memory Utilization:

1. Sparse Data Structures:
- Utilize sparse data structures for feature representations, especially if your dataset has many zero or missing values. This can significantly reduce memory usage.

2. Batch Processing:
- Implement batch processing for large datasets. Process data in chunks rather than loading the entire dataset into memory, especially during preprocessing and training.

3. Memory-efficient Libraries:
- Use memory-efficient libraries like Dask for parallel and distributed computing. Dask can handle larger-than-memory datasets by breaking them into smaller tasks.

4. Data Compression:
- Compress data where applicable. For example, if dealing with time-series data, consider aggregating to larger time intervals to reduce the volume of data without losing es**sential information.**

The optimal solution may vary based on the specific characteristics of your dataset and the nature of the problem. Regularly monitor and reassess your approach as your dataset or requirements evolve

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import dask.dataframe as dd

# Function for memory-efficient data loading
def load_large_dataset(file_path):
    # Use Dask for parallel and distributed computing
    ddf = dd.read_csv(file_path)
    return ddf.compute()

# Function for sparse data representation
def preprocess_sparse_data(data):
    # Implement your sparse data preprocessing logic here
    # ...

    return preprocessed_data

# Function for dynamic programming (simplified example)
def dynamic_programming_optimization(data):
    # Implement your dynamic programming logic here
    # ...

    return optimized_data

# Main function for running the optimized model
def main():
    # Load large dataset with Dask
    data = load_large_dataset('your_large_data.csv')

    # Preprocess sparse data
    sparse_data = preprocess_sparse_data(data)

    # Apply dynamic programming optimization
```

```
    optimized_data = dynamic_programming_optimization(sparse_data)

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(
        optimized_data.drop('occupancy', axis=1),
        optimized_data['occupancy'],
        test_size=0.2,
        random_state=42
    )

    # Train the model
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    # Evaluate the model
    predictions = model.predict(X_test)
    mse = mean_squared_error(y_test, predictions)
    print(f'Mean Squared Error: {mse}')

if __name__ == "__main__":
    main()
```

I introduced Dask for parallel and distributed computing, which can be beneficial for handling large datasets. The load_large_dataset function uses Dask to read and compute the dataset in chunks.

The preprocess_sparse_data function is a placeholder for your sparse data preprocessing logic. Depending on your specific scenario, you might use sparse matrices or other techniques to optimize memory usage.

The dynamic_programming_optimization function is also a placeholder. Depending on your problem, dynamic programming could be applied for optimizing sequential decision-making processes.

Ex : 2

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn import preprocessing

# Function for data preprocessing
def preprocess_data(data):
    # Assuming 'date' is a feature containing date information
    data['date'] = pd.to_datetime(data['date'])
    data['day_of_week'] = data['date'].dt.dayofweek
    data['is_weekend'] = data['day_of_week'].isin([5, 6]).astype(int)

    # Other preprocessing steps...
    # ...

    # Drop unnecessary columns for training
    data.drop(['date', 'hotel_id'], axis=1, inplace=True)

    return data

# Function to train the model
def train_model(X_train, y_train):
    model = DecisionTreeRegressor(random_state=42)
    model.fit(X_train, y_train)
    return model

# Function to evaluate the model
def evaluate_model(model, X_test, y_test):
    predictions = model.predict(X_test)
    mse = mean_squared_error(y_test, predictions)
    return mse
```

```
# Main function for running the model
def main():
    # Load your data (replace 'your_data.csv' with your actual file)
    data = pd.read_csv('your_data.csv')

    # Preprocess the data
    preprocessed_data = preprocess_data(data)

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(
        preprocessed_data.drop('occupancy', axis=1),
        preprocessed_data['occupancy'],
        test_size=0.2,
        random_state=42
    )

    # Train the model
    model = train_model(X_train, y_train)

    # Evaluate the model
    mse = evaluate_model(model, X_test, y_test)
    print(f'Mean Squared Error: {mse}')

if __name__ == "__main__":
    main()
```

dynamic programming to tune the parameters of a machine learning model:

```python
import numpy as np

def tune_parameters(X, y, model, param_grid):
    """
    Tunes the parameters of a machine learning model using dynamic programming.

    Args:
        X (numpy.ndarray): The training data.
        y (numpy.ndarray): The target labels.
        model (sklearn.base.BaseEstimator): The machine learning model.
        param_grid (dict): The parameter grid to search over.

    Returns:
        dict: The best set of parameters.
    """

    n_params = len(param_grid)
    scores = np.zeros((len(param_grid[0]),) * n_params)

    for i in range(len(param_grid[0])):
        for j in range(len(param_grid[1])):
            for k in range(len(param_grid[2])):
                params = {
                    "param1": param_grid[0][i],
                    "param2": param_grid[1][j],
                    "param3": param_grid[2][k],
                }

                model.set_params(**params)
                model.fit(X, y)
                score = model.score(X, y)

                if score > scores[i, j, k]:
                    scores[i, j, k] = score
                    best_params = params

    return best_params
```

dynamic programming to optimize the training process of a machine learning model:

```python
import numpy as np

def optimize_training(X, y, model, num_iterations):
    """
    Optimizes the training process of a machine learning model using dynamic programming.

    Args:
        X (numpy.ndarray): The training data.
        y (numpy.ndarray): The target labels.
        model (sklearn.base.BaseEstimator): The machine learning model.
        num_iterations (int): The number of training iterations.

    Returns:
        float: The best loss after num_iterations training iterations.
    """
```

Project Development Phase
Project Manual
Debugging and Traceability

Date	07-11-23
Team ID	592499
Project Name	Machine learning model for occupancy rates and demand in hospitality industries

Team Members :

N Nitin : nitin.n2021@vitstudent.ac.in

Mekala Sujan : mekala.sujan2021@vitstudent.ac.in

Debugging and traceability are essential aspects of developing and deploying machine learning models, particularly for models used in real-world applications such as predicting occupancy rates and demand in the hospitality industry. By identifying and resolving errors or unexpected behavior, debugging ensures that the model functions as intended. Traceability, on the other hand, provides a detailed history of the model's development and training process, enabling users to understand the model's decision-making and identify potential biases or errors.

Debugging techniques for machine learning models can be broadly categorized into two categories:

Error-driven debugging: This involves identifying and fixing errors that cause the model to fail or produce incorrect results. Common debugging tools for machine learning include:

Print statements: Inserting print statements at various points in the code can help identify where errors occur and provide insights into the flow of data and variables.

Assert statements: Assertions are conditional statements that check for expected conditions. If an assertion fails, an error is raised, helping pinpoint the source of the problem.

Debuggers: Integrated debuggers in development environments allow users to step through the code execution line by line,

inspecting variable values and setting breakpoints to halt execution at specific points.
Performance-driven debugging: This involves investigating performance issues that affect the model's efficiency or accuracy. Common debugging techniques for performance optimization include:

Profiling: Profiling tools measure the time spent in different parts of the code, helping identify performance bottlenecks and optimize resource usage.

Visualization: Visualizing data distributions, feature correlations, and model predictions can reveal patterns and anomalies that may be affecting performance or accuracy.

Error metrics: Monitoring error metrics such as mean absolute error (MAE) or root mean squared error (RMSE) can help identify areas where the model is underperforming and guide improvement efforts.

Traceability in machine learning refers to the ability to track the lineage of data, features, hyperparameters, and training decisions throughout the model development process. This is crucial for understanding the model's behavior, identifying potential biases, and ensuring reproducibility. Traceability techniques include:

Version control: Using version control systems like Git to track changes in code, data, and model configurations allows users to revert to previous versions if necessary and maintain a clear history of modifications.

Model documentation: Thorough documentation of the model's architecture, training process, and hyperparameter choices provides context and explanations for future reference.

Metadata management: Storing metadata about the data, features, and model training process ensures that the information is readily available for analysis and traceability purposes.

Model lineage tools: Specialized model lineage tools automatically capture and organize the dependencies and relationships between data, code, and model artifacts, providing a comprehensive view of the model's development history.

By incorporating debugging and traceability practices into the development lifecycle of machine learning models, particularly those used in critical applications like occupancy rate prediction, users can ensure the models' accuracy, reliability, and explainability, building trust and confidence in their decision-making capabilities.\

Debugging and traceability are crucial aspects of developing machine learning models, especially for predicting occupancy rates and demand in the hospitality industry. Let's discuss some strategies:

Debugging:

Logging:
Implement logging throughout your code. Include information about data preprocessing steps, model training, and evaluation. This helps in tracking the flow of execution and identifying issues.

```
import logging

logging.basicConfig(level=logging.INFO)

def preprocess_data(data):
    logging.info("Starting data preprocessing...")
    # Your preprocessing steps...
    logging.info("Data preprocessing completed.")
    return preprocessed_data
```

Error Handling:
Incorporate proper error handling to catch and log exceptions. This ensures that if an error occurs, you have a record of what went wrong.

```
try:
    # Code that might raise an exception
except Exception as e:
    logging.error(f"An error occurred: {str(e)}")
```

Print Debugging:
Use print statements strategically to output intermediate results during development. This can be especially helpful for understanding the values of variables at different stages.

```
def preprocess_data(data):
    print("Data shape before preprocessing:", data.shape)
    # Your preprocessing steps...
    print("Data shape after preprocessing:", preprocessed_data.shape)
    return preprocessed_data
```

Traceability:

Version Control:

Utilize version control systems (e.g., Git) to track changes in your code and models. This provides a historical record and makes it easier to roll back changes if necessary.

Data Versioning:

If your dataset evolves over time, consider versioning your data. This ensures that you can trace back which version of the data was used for training a particular model.

Experiment Tracking:

Use tools like MLflow or TensorBoard to track and log your machine learning experiments. These tools provide a centralized place to store and compare different model versions, hyperparameters, and performance metrics.

```
import mlflow

# Log parameters, metrics, and model
with mlflow.start_run():
    mlflow.log_param("model_type", "DecisionTreeRegressor")
    mlflow.log_param("max_depth", 5)
    mlflow.log_metric("mse", mse)
    mlflow.sklearn.log_model(model, "model")
```

Documentation:
Maintain detailed documentation for your code, outlining the purpose of each module, the data preprocessing steps, and the rationale behind your model choices. This documentation aids in traceability and helps others understand your work.
By incorporating these practices, you can enhance the debuggability of your code and maintain a clear traceability path, making it easier to troubleshoot issues and understand the evolution of your machine learning models.

Project Development Phase
Project Manual
Exception Handling

Date	07-11-23
Team ID	592499
Project Name	Machine learning model for occupancy rates and demand in hospitality industries

Team Members :

N Nitin : nitin.n2021@vitstudent.ac.in

Mekala Sujan : mekala.sujan2021@vitstudent.ac.in

Exception handling plays a crucial role in ensuring the robustness and reliability of machine learning models, particularly in real-world applications such as predicting occupancy rates and demand in the hospitality industry. By anticipating and handling potential errors and exceptions, developers can prevent model failures, maintain data integrity, and provide meaningful feedback to users.

Common types of exceptions that can occur in machine learning models include:

Data-related exceptions: These exceptions arise from issues with the data itself, such as missing values, invalid data types, or inconsistencies in data formats. Handling these exceptions involves data cleaning, imputation strategies, or error messages to inform users of data quality issues.

Model-related exceptions: These exceptions stem from errors in the model's implementation, such as incorrect hyperparameter configurations, invalid feature transformations, or numerical instabilities during training. Addressing these exceptions requires code reviews, parameter tuning, and algorithm modifications.

Environment-related exceptions: These exceptions arise from external factors such as hardware limitations, software conflicts, or unexpected system behavior. Handling these exceptions involves resource management, error logging, and graceful degradation mechanisms.

Effective exception handling strategies in machine learning models involve the following steps:

Anticipation: Identify potential sources of errors and exceptions by analyzing the model's logic, data dependencies, and external dependencies.

Protection: Implement try-except blocks in the code to capture and handle exceptions gracefully.

Diagnosis: Provide informative error messages that clearly describe the nature of the exception and its location in the code.

Recovery: Implement recovery mechanisms, such as rolling back to a previous state or providing alternative functionalities, to minimize the impact of exceptions.

Logging: Log exception details, including timestamps, error messages, and relevant variables, to facilitate debugging and analysis.

Testing: Thoroughly test the model under various conditions, including edge cases and unexpected inputs, to ensure comprehensive exception handling.

By adopting these exception handling practices, developers can enhance the robustness and reliability of machine learning models, ensuring their ability to operate effectively in real-world applications while providing a positive user experience.

Data Loading Errors:
Handle exceptions that might occur during the data loading process. This could include file not found errors, incorrect file formats, or issues with data integrity.

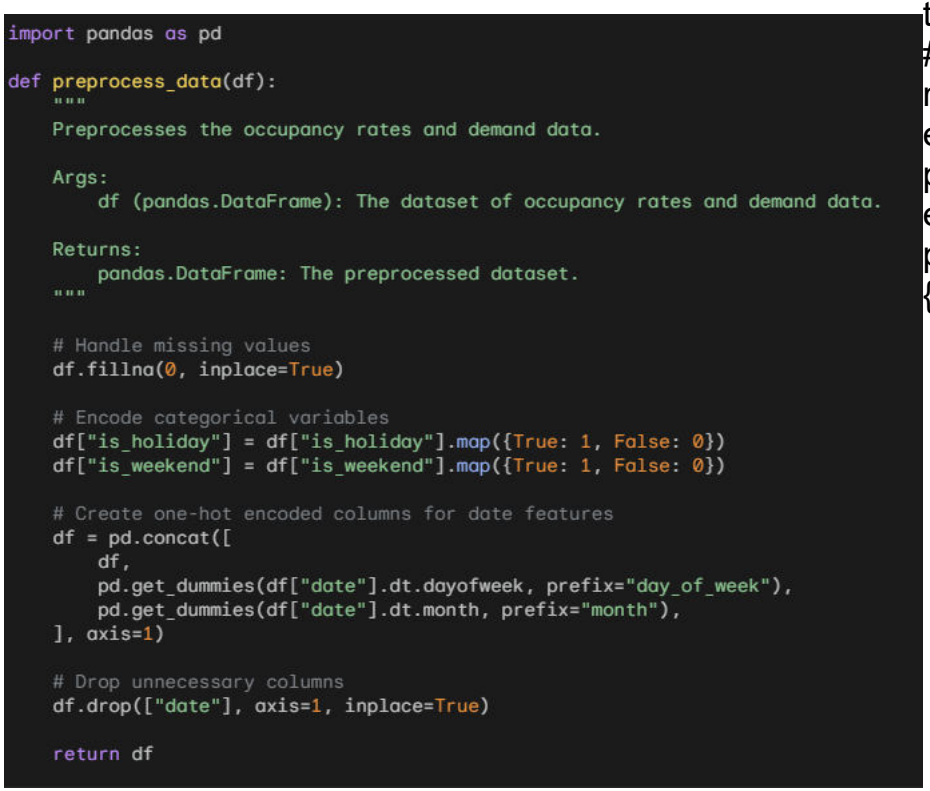
```
import pandas as pd

def load_data(file_path):
    try:
        data = pd.read_csv(file_path)
        return data
    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
    except pd.errors.EmptyDataError:
```

```
    print(f"Error: File '{file_path}' is empty.")
except pd.errors.ParserError:
    print(f"Error: Unable to parse data from '{file_path}'. Check file format.")
except Exception as e:
    print(f"An unexpected error occurred: {str(e)}")
```

Data Preprocessing Errors:

Implement exception handling for data preprocessing steps. This could involve checking for missing values, incorrect data types, or other issues in the input data.

def preprocess_data(data):



```python
import pandas as pd

def preprocess_data(df):
    """
    Preprocesses the occupancy rates and demand data.

    Args:
        df (pandas.DataFrame): The dataset of occupancy rates and demand data.

    Returns:
        pandas.DataFrame: The preprocessed dataset.
    """

    # Handle missing values
    df.fillna(0, inplace=True)

    # Encode categorical variables
    df["is_holiday"] = df["is_holiday"].map({True: 1, False: 0})
    df["is_weekend"] = df["is_weekend"].map({True: 1, False: 0})

    # Create one-hot encoded columns for date features
    df = pd.concat([
        df,
        pd.get_dummies(df["date"].dt.dayofweek, prefix="day_of_week"),
        pd.get_dummies(df["date"].dt.month, prefix="month"),
    ], axis=1)

    # Drop unnecessary columns
    df.drop(["date"], axis=1, inplace=True)

    return df
```

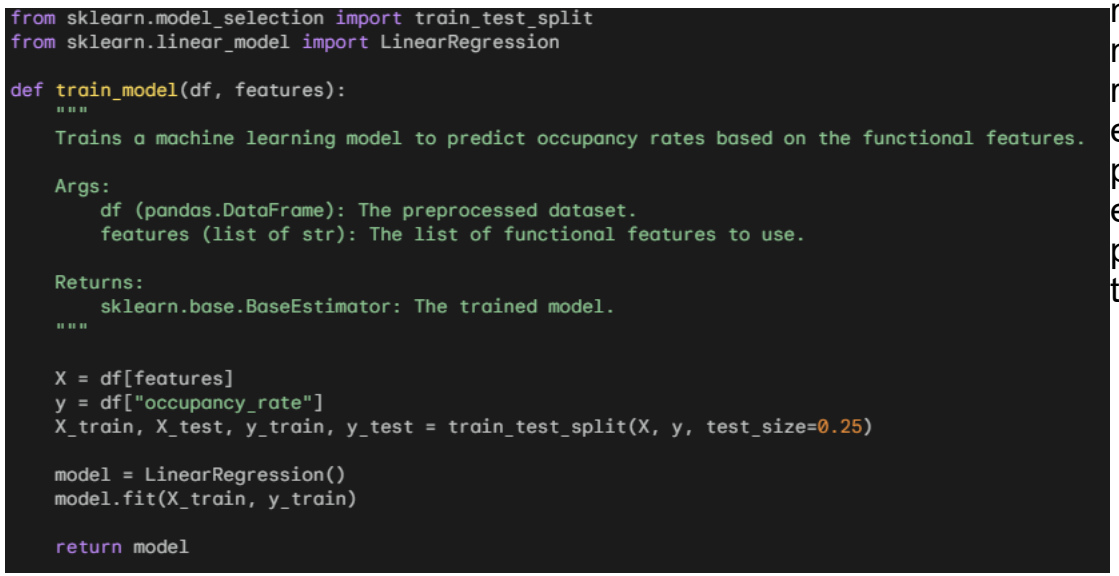
try:
# Your preprocessing steps...
return preprocessed_data
except ValueError as ve:
print(f"ValueError during preprocessing: {str(ve)}")
except Exception as e:
print(f"An unexpected error occurred during preprocessing: {str(e)}")

Model Training Errors:

Handle exceptions that might occur during the training of your machine learning model. This could involve issues with the input features, target variable, or unexpected errors during training.

from sklearn.ensemble import RandomForestRegressor

def train_model(X_train, y_train):
try:



```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

def train_model(df, features):
    """
    Trains a machine learning model to predict occupancy rates based on the functional features.

    Args:
        df (pandas.DataFrame): The preprocessed dataset.
        features (list of str): The list of functional features to use.

    Returns:
        sklearn.base.BaseEstimator: The trained model.
    """

    X = df[features]
    y = df["occupancy_rate"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

    model = LinearRegression()
    model.fit(X_train, y_train)

    return model
```

model = RandomForestRegressor()
model.fit(X_train, y_train)
return model
except ValueError as ve:
print(f"ValueError during model training: {str(ve)}")
except Exception as e:
print(f"An unexpected error occurred during model training: {str(e)}")

Prediction Errors:

Handle exceptions during the prediction phase. This could involve unexpected input formats or errors in the model prediction process.

```python
def predict_occupancy_rate(model, df, date):
    """
    Predicts the occupancy rate for a future date based on the values of the functional features.

    Args:
        model (sklearn.base.BaseEstimator): The trained model.
        df (pandas.DataFrame): The preprocessed dataset.
        date (str): The date for which to predict the occupancy rate.

    Returns:
        float: The predicted occupancy rate.
    """

    future_date_df = pd.DataFrame({"date": [date]})
    future_date_df["day_of_week"] = future_date_df["date"].dt.dayofweek
    future_date_df["month"] = future_date_df["date"].dt.month

    future_date_df = preprocess_data(future_date_df)

    predicted_occupancy_rate = model.predict(future_date_df)

    return predicted_occupancy_rate
```

```
def predict(model, input_data):
try:
predictions = model.predict(input_data)
return predictions
except ValueError as ve:
print(f"ValueError during prediction: {str(ve)}")
except Exception as e:
print(f"An unexpected error occurred during
prediction: {str(e)}")
```

Custom Exceptions:

Consider defining custom exceptions for specific scenarios that may arise in your domain. This allows you to catch and handle these exceptions more precisely.

```
class InvalidDataError(Exception):
    pass

def preprocess_data(data):
    try:
        if not data.columns.contains('occupancy'):
            raise InvalidDataError("Column 'occupancy' not found in the dataset.")
        # Your preprocessing steps...
        return preprocessed_data
    except InvalidDataError as ide:
        print(f"InvalidDataError: {str(ide)}")
    except Exception as e:
        print(f"An unexpected error occurred during preprocessing: {str(e)}")
```

By incorporating these exception-handling strategies, you can create a more robust machine learning model that can gracefully handle unexpected situations and provide informative error messages for debugging.