Grace Samuel

Preliminary Results

Step 1 – Perform the data processing steps proposed in the Analytic Plan

Data Processing ---
According to the Analytic Plan, all variables can be put into four categories: Exclude, Alter, Standardize, and Leave as is. See below for breakdown.

Exclude: Booking_ID

Alter: type_of_meal_plan, room_type_reserved, arrival_date, market_segment_type, booking_status

Standardize: no_of_adults, no_of_children, no_of_weekend_nights, no_of_week_nights, lead_time, no_of_previous_cancellations, no_of_previous_bookings_not_canceled, avg_price_per_room, no_of_special_requests

Leave: required_car_parking_space, repeated_guest

The Analytics Plan also handles missing, unknown, and NA values.

Below is the executed plan:
Load Libraries ---

```
# Load libraries

import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

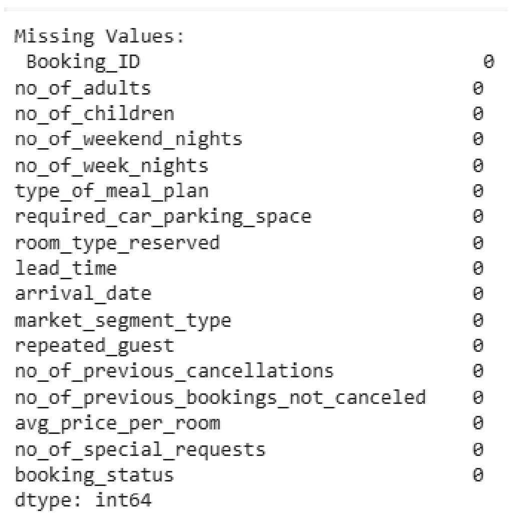
Load Data ---

```
# Load data
project_data = pd.read_csv('/content/project_data.csv')
```

Check for Missing Values ---

```
# Check for missing values
missing_values = project_data.isnull().sum()
print("Missing Values:\n", missing_values)
```

See below for output. No missing values, so no further action needed.



```
Missing Values:
 Booking_ID                              0
no_of_adults                            0
no_of_children                          0
no_of_weekend_nights                    0
no_of_week_nights                       0
type_of_meal_plan                       0
required_car_parking_space              0
room_type_reserved                      0
lead_time                               0
arrival_date                            0
market_segment_type                     0
repeated_guest                          0
no_of_previous_cancellations            0
no_of_previous_bookings_not_canceled    0
avg_price_per_room                      0
no_of_special_requests                  0
booking_status                          0
dtype: int64
```

Exclude Columns ---
Below, I will exclude the Booking_ID column because it is a unique identifier and not useful to the analysis.

```
# Exclude Booking_ID
project_data.drop(columns=['Booking_ID'], inplace=True)
```

Alter Columns ---
Below, I will one-hot encode for categorical variables so the category types can be used in analysis.

```
# One-hot Encode categorical variables
categorical_cols = ["type_of_meal_plan", "room_type_reserved",
"market_segment_type"]
project_data = pd.get_dummies(project_data, columns=categorical_cols,
drop_first=True)
```

Below, I will transform arrival_date into day, month, and year since each of those, individually, can influence cancellations. I will then one-hot encode day and month since they are categorical. I will not one-hot encode year, as that would create unnecessary columns. Instead, I will standardize arrival_year.

```
# Transform arrival_date into day of week, month, and year
project_data["arrival_date"] =
pd.to_datetime(project_data["arrival_date"])
project_data["arrival_day_of_week"] =
project_data["arrival_date"].dt.dayofweek
project_data["arrival_month"] = project_data["arrival_date"].dt.month
project_data["arrival_year"] = project_data["arrival_date"].dt.year
project_data.drop(columns=["arrival_date"], inplace=True)
```

```
# One-hot encode day of week and month
project_data = pd.get_dummies(project_data,
columns=["arrival_day_of_week", "arrival_month"], drop_first=True)
```

Standardize Columns --

Below, I will standardize numerical variables so they have a mean of 0 and standard deviation of 1 and contribute to the model proportionally.

```
# Standardize numerical variables
num_cols = [
    "no_of_adults", "no_of_children", "no_of_weekend_nights",
"no_of_week_nights",
    "lead_time", "no_of_previous_cancellations",
"no_of_previous_bookings_not_canceled",
    "avg_price_per_room", "no_of_special_requests", "arrival_year"
]
scaler = StandardScaler()
project_data[num_cols] = scaler.fit_transform(project_data[num_cols])
```

Transform Target Variable ---

```
# Transform target variable into binary
project_data["booking_status"] =
project_data["booking_status"].map({"not_canceled": 0, "canceled": 1})
```

Convert True/False to 0s and 1s ---

```
# Convert True/False to 0s and 1s
bool_cols = project_data.select_dtypes(include=['bool']).columns
project_data[bool_cols] = project_data[bool_cols].astype(int)
```

After this, I saved the processed data as a new file to be used in the following steps.

Step 2 – Train a dense neural network model to predict the label.

Prepare Features --

```
# Import libraries
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset
```

```
# Load processed data
df = pd.read_csv("/content/processed_data.csv")
```

```
# Separate features (X) and target variable (y)
X = df.drop(columns=["booking_status"]).values
y = df["booking_status"].values
```

```python
# Convert to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32).view(-1, 1)
```

```python
# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_tensor, y_tensor,
test_size=0.2, random_state=42)
```

```python
# Create PyTorch DataLoader objects for batch training
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Define the Dense Neural Network --

```python
# Define a feedforward neural network
class DenseNN(nn.Module):
    def __init__(self, input_size):
        super(DenseNN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size, 64),
            nn.ReLU(),

            nn.Linear(64, 32),
            nn.ReLU(),

            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

# Initialize model
input_size = X_train.shape[1]
model = DenseNN(input_size)
```

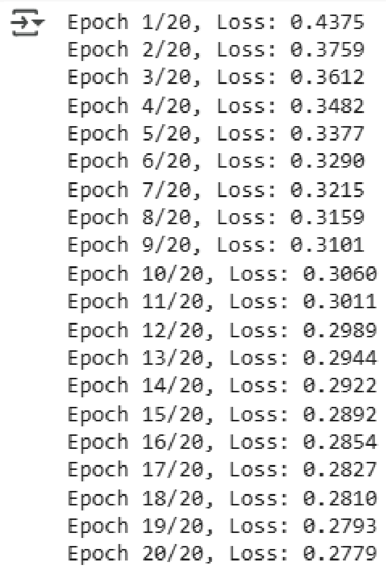
Define Loss Function & Optimizer ---

```python
# Binary Cross-Entropy Loss
criterion = nn.BCELoss()

# Adam Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Train the Model --

The output below shows a decreasing training loss.

```python
# Training loop
epochs = 20
for epoch in range(epochs):
    model.train()
    total_loss = 0

    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{epochs}, Loss:
{total_loss/len(train_loader):.4f}")
```



```
Epoch 1/20, Loss: 0.4375
Epoch 2/20, Loss: 0.3759
Epoch 3/20, Loss: 0.3612
Epoch 4/20, Loss: 0.3482
Epoch 5/20, Loss: 0.3377
Epoch 6/20, Loss: 0.3290
Epoch 7/20, Loss: 0.3215
Epoch 8/20, Loss: 0.3159
Epoch 9/20, Loss: 0.3101
Epoch 10/20, Loss: 0.3060
Epoch 11/20, Loss: 0.3011
Epoch 12/20, Loss: 0.2989
Epoch 13/20, Loss: 0.2944
Epoch 14/20, Loss: 0.2922
Epoch 15/20, Loss: 0.2892
Epoch 16/20, Loss: 0.2854
Epoch 17/20, Loss: 0.2827
Epoch 18/20, Loss: 0.2810
Epoch 19/20, Loss: 0.2793
Epoch 20/20, Loss: 0.2779
```

Evaluate Model Performance --

The output below shows the model achieved 86.16%.

```python
# Evaluate on test set
model.eval()
correct, total = 0, 0

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        y_pred = model(X_batch)
        predicted = (y_pred > 0.5).float()
        correct += (predicted == y_batch).sum().item()
        total += y_batch.size(0)

accuracy = correct / total
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Test Accuracy: 0.8616
```

Step 3 – Evaluate the neural network using learning curves on training and validation sets.

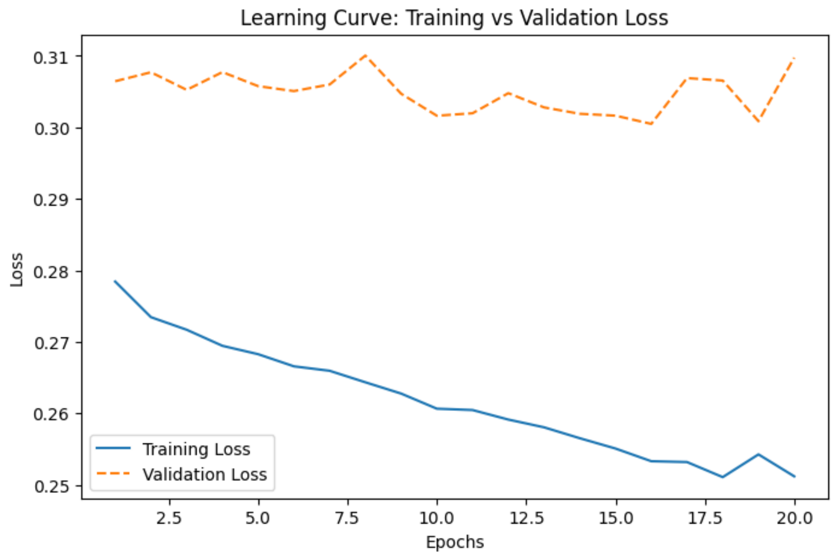
Track Losses --

```
import matplotlib.pyplot as plt

# Lists to store loss values
train_losses = []
val_losses = []

epochs = 20
for epoch in range(epochs):
    model.train()
    train_loss = 0

    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # Calculate average training loss
    avg_train_loss = train_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Validation phase (no gradient updates)
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for X_val, y_val in test_loader:
            y_pred = model(X_val)
            loss = criterion(y_pred, y_val)
            val_loss += loss.item()

    # Calculate average validation loss
    avg_val_loss = val_loss / len(test_loader)
    val_losses.append(avg_val_loss)
    print(f"Epoch {epoch+1}/{epochs}, Training Loss: {avg_train_loss:.4f},
Validation Loss: {avg_val_loss:.4f}")
```

```
Epoch 1/20, Training Loss: 0.2784, Validation Loss: 0.3065
Epoch 2/20, Training Loss: 0.2735, Validation Loss: 0.3077
Epoch 3/20, Training Loss: 0.2717, Validation Loss: 0.3052
Epoch 4/20, Training Loss: 0.2694, Validation Loss: 0.3077
Epoch 5/20, Training Loss: 0.2683, Validation Loss: 0.3058
Epoch 6/20, Training Loss: 0.2666, Validation Loss: 0.3051
Epoch 7/20, Training Loss: 0.2660, Validation Loss: 0.3060
Epoch 8/20, Training Loss: 0.2643, Validation Loss: 0.3100
Epoch 9/20, Training Loss: 0.2628, Validation Loss: 0.3047
Epoch 10/20, Training Loss: 0.2606, Validation Loss: 0.3016
Epoch 11/20, Training Loss: 0.2605, Validation Loss: 0.3020
Epoch 12/20, Training Loss: 0.2591, Validation Loss: 0.3048
Epoch 13/20, Training Loss: 0.2580, Validation Loss: 0.3028
Epoch 14/20, Training Loss: 0.2565, Validation Loss: 0.3019
Epoch 15/20, Training Loss: 0.2551, Validation Loss: 0.3016
Epoch 16/20, Training Loss: 0.2533, Validation Loss: 0.3005
Epoch 17/20, Training Loss: 0.2532, Validation Loss: 0.3069
Epoch 18/20, Training Loss: 0.2511, Validation Loss: 0.3066
Epoch 19/20, Training Loss: 0.2543, Validation Loss: 0.3009
Epoch 20/20, Training Loss: 0.2512, Validation Loss: 0.3097
```

Plot Learning Curves ---

```
plt.figure(figsize=(8, 5))
plt.plot(range(1, epochs + 1), train_losses, label="Training Loss")
plt.plot(range(1, epochs + 1), val_losses, label="Validation Loss",
linestyle="dashed")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Learning Curve: Training vs Validation Loss")
plt.show()
```



Evaluation --

The learning curves indicate mild overfitting, as the training loss steadily decreases while the validation loss stays relatively flat and fluctuates slightly. This suggests that while the model is capturing patterns in the training data, it is not generalizing well to new data. To improve this, Dropout layers will be added to prevent reliance on specific neurons, and L2 regularization will help keep weight updates in check. The learning rate will also be adjusted dynamically using a scheduler to control overfitting, and the loss function will be switched to BCEWithLogitsLoss() for better numerical stability. Additionally, early stopping will be used to stop training when validation loss stops improving, helping the model generalize more effectively.

Next Steps for the Final Report --

Based on the preliminary model's evaluation, the next steps in data processing and feature engineering will focus on improving generalization and overall performance. Feature selection will be used to remove less useful features and simplify the model. If needed, polynomial or interaction features may be introduced to capture more complex relationships. To handle class imbalance, techniques like oversampling, undersampling, or class-weighted loss functions will be considered. Scaling methods will also be reviewed to ensure consistency across features, and approaches like PCA or feature embeddings may be tested to enhance data representation. These adjustments will be evaluated in the Final Report to measure their impact on accuracy and overfitting.