Sample Report

Grace Samuel

This report investigates the ideal dense feed-forward neural network to identify hotel bookings with a high risk of cancellation. ABC Hotels provided a dataset containing information on over 35,000 bookings, where it is known whether each reservation was canceled or completed. The results of this report have implications for hotel revenue management, targeted promotional efforts, and occupancy forecasting.

In the Analytic Plan phase of the investigation, "booking_status" (0 = Not Canceled, 1 = Canceled) was named as the target variable. A plan was made for each feature in the dataset to delete, modify, or leave it as is. In the Preliminary phase, data cleaning and initial model training and evaluation took place. During data cleaning, missing values were handled appropriately, and categorical variables such as type_of_meal_plan and market_segment_type were one-hot encoded. Numerical features such as lead time and average price per room were standardized to ensure equal contribution to model learning. An initial dense feedforward neural network (Model #1) was trained and evaluated. Using the ROC Curve Comparison and AUC, it was determined that Model #1 performed well but required improvements to prevent underfitting and increase predictive power. This finding suggested the need for a more flexible and complex model architecture.

For the Final Report, two dense feedforward neural networks were trained and evaluated. Model #1 (a simple network with fewer hidden layers) and Model #2 (a deeper network with added batch normalization, dropout, and L2 regularization) were each assessed individually using ROC Curves, AUC, and Calibration Plots to determine if adjustments were needed. After these individual evaluations, a side-by-side comparison of ROC/AUC and Calibration Plots was conducted to identify the better-performing model. Model #2 was chosen as the recommended model because it had an AUC of 0.9352, the strongest ROC Curve, and effectively balanced complexity with generalization. The addition of regularization techniques in Model #2 helped reduce overfitting, making it a more reliable choice for predicting cancellations. Further analysis of the calibration plots confirmed that Model #2 was well-calibrated, meaning its predicted probabilities closely matched actual cancellation outcomes. This is particularly useful for ABC Hotels, as it allows them to confidently use probability scores to target high-risk bookings with strategic promotions and interventions.

The next step for ABC Hotels will be to use the predictive model to implement strategies that reduce cancellations. For example, hotel management can offer discounts on non-refundable rates to customers with a high predicted probability of cancellation, encouraging them to keep their reservations. Personalized reminders or exclusive upgrade offers can be sent to customers identified as high-risk to incentivize them to show up for their booking. If the model predicts a low cancellation probability, ABC Hotels can divert the funds they would have spent on targeted advertising elsewhere.

It is recommended that further investigations be conducted to better understand why certain bookings are more likely to be canceled. Are customers canceling due to price, other options, or flexible cancellation policies? Do specific market segments (e.g., families, people traveling for work) exhibit higher cancellation rates? Are factors such as special requests or type of meal plan influencing cancellations? Since booking cancellations have a direct impact on revenue forecasting and operational planning, gaining deeper insights into cancellation reasons is important for optimizing hotel performance. By incorporating Model #2 into their decision-making process, ABC Hotels can reduce cancellations, increase revenue, and improve customer retention strategies.

CODE AND VISUALIZATIONS

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.

MODEL #1

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
```

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

```
# 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)
```

```
# 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 ---

```
# 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 ---

```
# 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.

```
# 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%.

```
# 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
```

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:
```

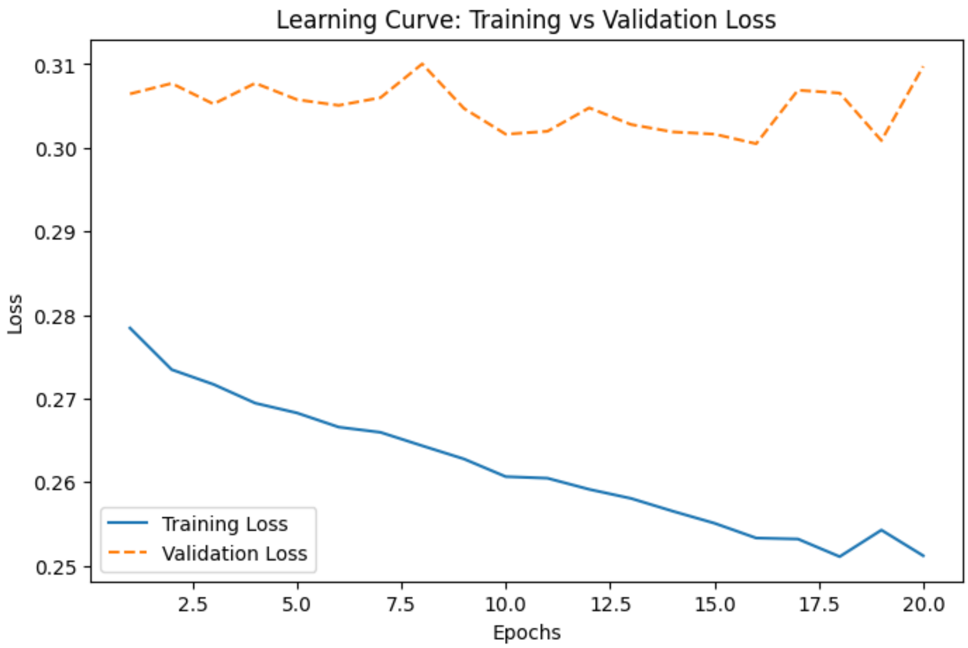
```
        optimizer.zero_grad()
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # 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")
```

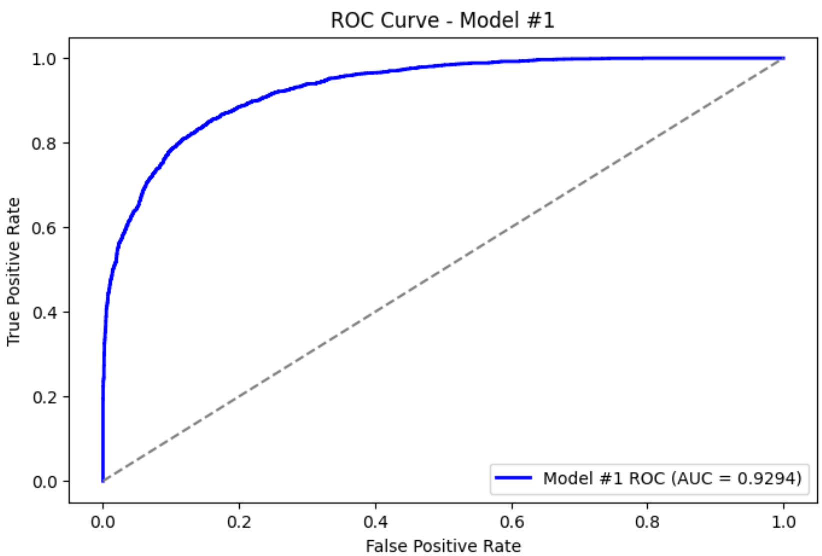
```
plt.plot(range(1, epochs + 1), val_losses, label="Validation
Loss", linestyle="dashed")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Learning Curve: Training vs Validation Loss")
plt.show()
```



ROC/AUC

```
# Set Model #1 to evaluation mode
model.eval()
y_scores_model1 = []

# Get predictions on the test set
with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model(X_batch)
        y_scores_model1.extend(y_pred.cpu().numpy())

# Convert true labels to NumPy
y_true = y_test.cpu().numpy().flatten()
y_scores_model1 = np.array(y_scores_model1).flatten()

# Compute ROC curve and AUC score
fpr1, tpr1, _ = roc_curve(y_true, y_scores_model1)
roc_auc1 = auc(fpr1, tpr1)
```

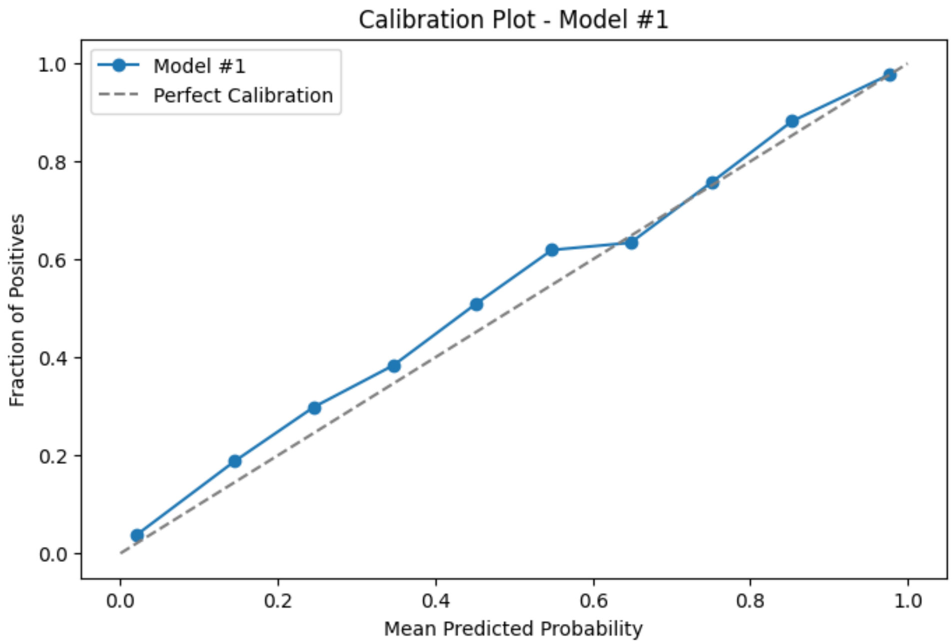
```python
# Plot ROC Curve
plt.figure(figsize=(8, 5))
plt.plot(fpr1, tpr1, color='blue', lw=2, label=f'Model #1 ROC
(AUC = {roc_auc1:.4f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle="dashed")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Model #1")
plt.legend(loc="lower right")
plt.show()
```



Calibration Plot ---

```python
# Set Model #1 to evaluation mode
model.eval()
y_scores_model1 = []

# Get predicted probabilities on the test set
with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model(X_batch)
        y_scores_model1.extend(y_pred.cpu().numpy())

# Convert true labels to NumPy
y_true = y_test.cpu().numpy().flatten()
y_scores_model1 = np.array(y_scores_model1).flatten()

# Compute calibration curve
fraction_of_positives, mean_predicted_value =
calibration_curve(y_true, y_scores_model1, n_bins=10)
```

```python
# Plot Calibration Curve for Model #1
plt.figure(figsize=(8, 5))
plt.plot(mean_predicted_value, fraction_of_positives,
marker="o", label="Model #1")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray",
label="Perfect Calibration")
plt.xlabel("Mean Predicted Probability")
plt.ylabel("Fraction of Positives")
plt.title("Calibration Plot - Model #1")
plt.legend()
plt.show()
```



MODEL # 2

Define the Dense Neural Network --

```python
# Define the improved feedforward neural network with Dropout
class DenseNN2(nn.Module):
    def __init__(self, input_size):
        super(DenseNN2, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.BatchNorm1d(128),
            nn.LeakyReLU(),
            nn.Dropout(0.3),

            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
```

```
            nn.LeakyReLU(),
            nn.Dropout(0.3),

            nn.Linear(64, 32),
            nn.BatchNorm1d(32),
            nn.LeakyReLU(),
            nn.Dropout(0.3),

            nn.Linear(32, 16),
            nn.LeakyReLU(),

            nn.Linear(16, 1),
            nn.Sigmoid()
        )

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

# Initialize Model #2
input_size = X_train.shape[1]
model2 = DenseNN2(input_size)
```

Define Loss Function & Optimizer ---

```
# Define Loss Function & Optimizer with L2 Regularization
criterion = nn.BCELoss()
optimizer = optim.Adam(model2.parameters(), lr=0.001,
weight_decay=1e-4)
```

Train the Model ---

```
# Early Stopping Setup
early_stopping_patience = 10
best_val_loss = float("inf")
patience_counter = 0
```

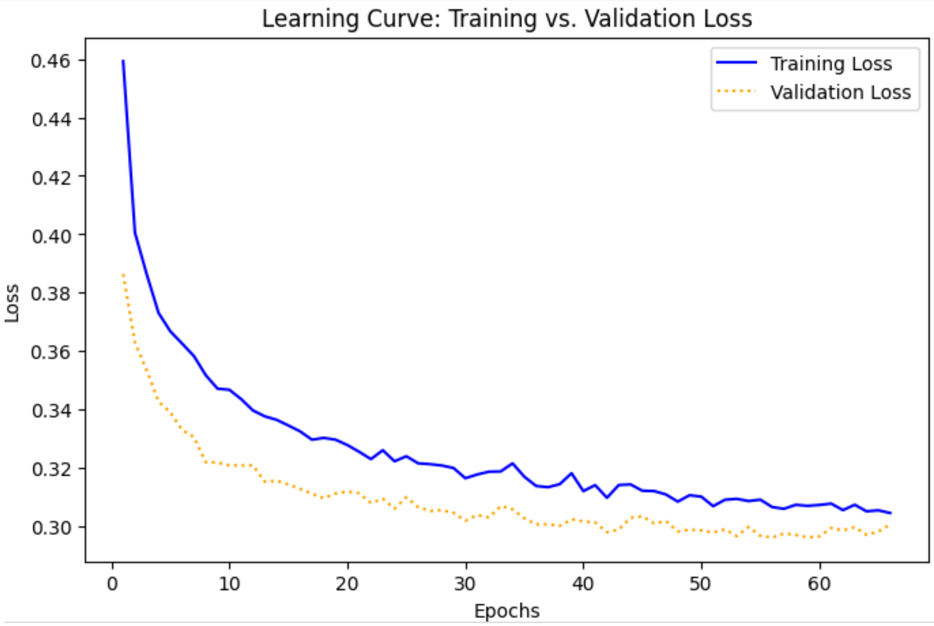
```
# Training Loop with Early Stopping
epochs = 2000
train_losses = []
val_losses = []

for epoch in range(epochs):
    model2.train()
```

```
    train_loss = 0

    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = model2(X_batch)
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    avg_train_loss = train_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Validation Phase
    model2.eval()
    val_loss = 0
    with torch.no_grad():
        for X_val, y_val in test_loader:
            y_pred = model2(X_val)
            loss = criterion(y_pred, y_val)
            val_loss += loss.item()

    avg_val_loss = val_loss / len(test_loader)
    val_losses.append(avg_val_loss)

    # Print loss every 100 epochs
    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Training Loss:
{avg_train_loss:.4f}, Validation Loss: {avg_val_loss:.4f}")

    # Early Stopping Check
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= early_stopping_patience:
        print(f"Early stopping triggered at epoch {epoch+1}")
        break
```

```
Early stopping triggered at epoch 66
```

Plot Learning Curves --

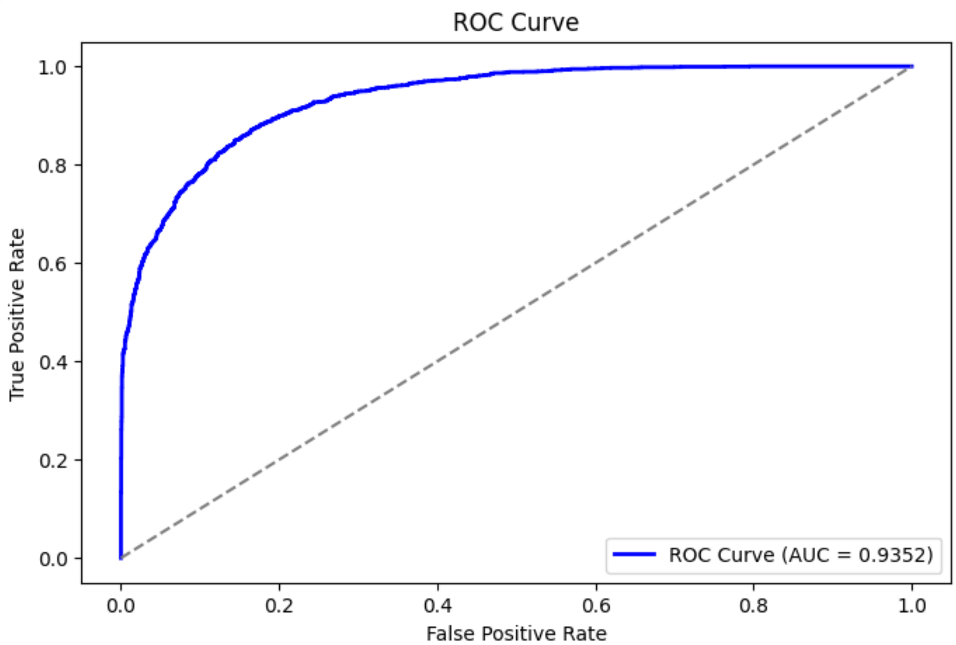
```
# Plot Training and Validation Loss
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(train_losses) + 1), train_losses,
label="Training Loss", color='blue')
plt.plot(range(1, len(val_losses) + 1), val_losses,
label="Validation Loss", color='red', linestyle="dashed")

# Add labels and title
plt.xlabel("Epochs")
plt
```



ROC/AUC ---

```
# Get model predictions on the test set
model2.eval()
y_scores = []

with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model2(X_batch)
        y_scores.extend(y_pred.cpu().numpy())

# Convert true labels to NumPy
y_true = y_test.cpu().numpy().flatten()
y_scores = np.array(y_scores).flatten()

# Compute ROC curve and AUC score
fpr, tpr, _ = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)
```

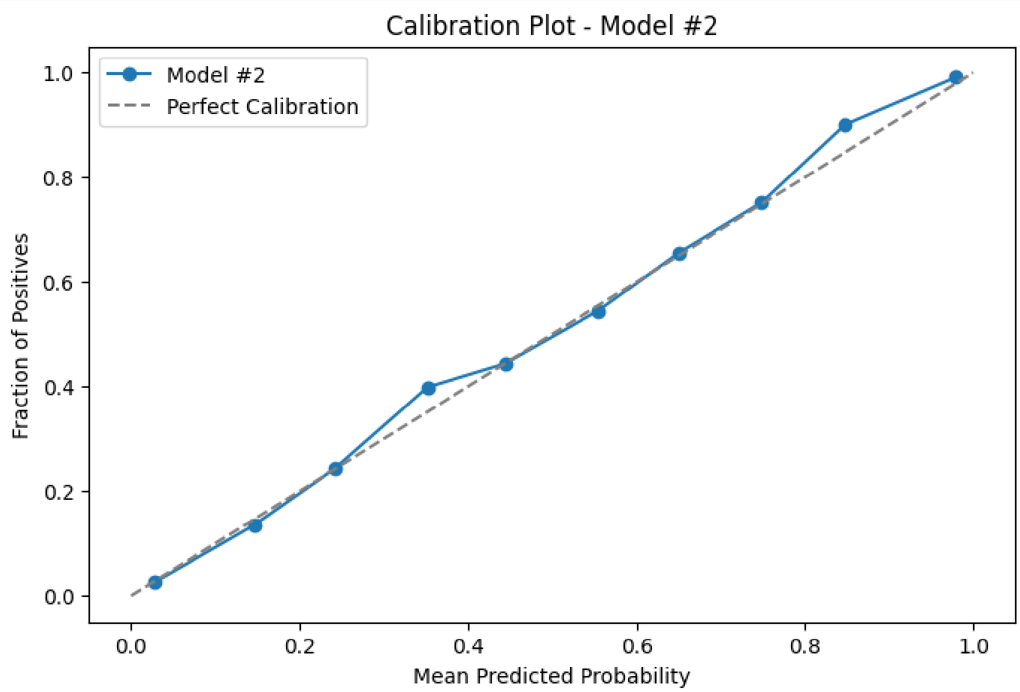
```
# Plot ROC Curve
plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC Curve (AUC =
{roc_auc:.4f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle="dashed")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend(loc="lower right")
plt.show()
```



Calibration Plot ---

```
# Get model predictions on the test set
model2.eval()
y_scores = []

with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model2(X_batch)
        y_scores.extend(y_pred.cpu().numpy())

# Convert true labels to NumPy
y_true = y_test.cpu().numpy().flatten()
y_scores = np.array(y_scores).flatten()

# Compute calibration curve
```

```python
fraction_of_positives, mean_predicted_value =
calibration_curve(y_true, y_scores, n_bins=10)

# Plot Calibration Curve
plt.figure(figsize=(8, 5))
plt.plot(mean_predicted_value, fraction_of_positives,
marker="o", label="Model #2")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray",
label="Perfect Calibration")
plt.xlabel("Mean Predicted Probability")
plt.ylabel("Fraction of Positives")
plt.title("Calibration Plot - Model #2")
plt.legend()
plt.show()
```

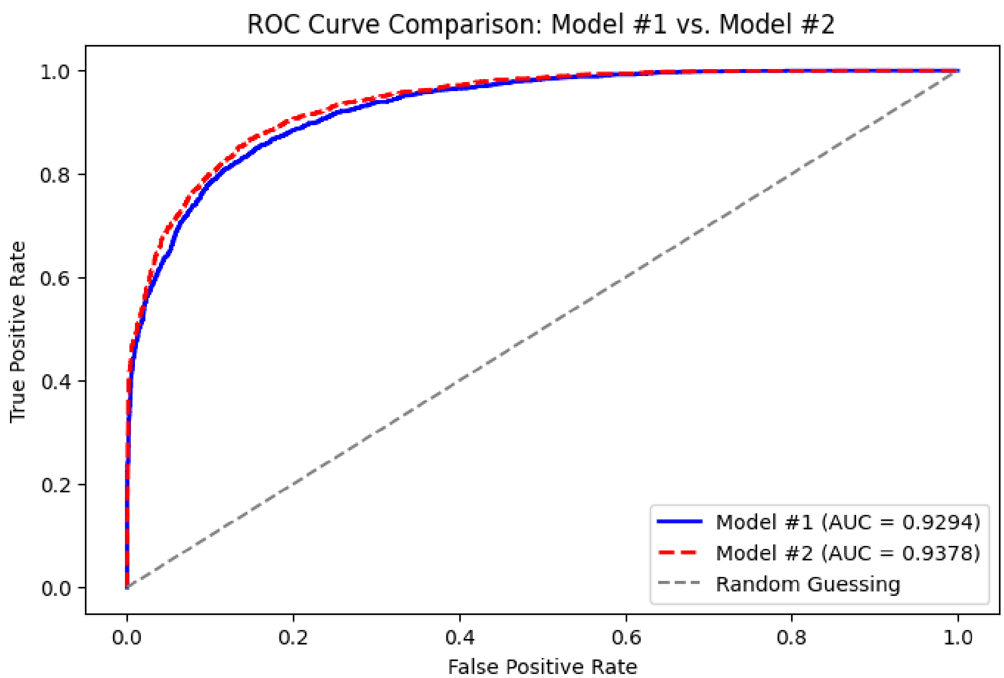


Side by Side Comparisons – Models 1 & 2

Combined ROC/AUC Plots ---

```python
# Make sure both models are in evaluation mode
model.eval()
model2.eval()

# Get predictions for Model #1
y_scores_model1 = []
with torch.no_grad():
```

```
    for X_batch, _ in test_loader:
        y_pred = model(X_batch)
        y_scores_model1.extend(y_pred.cpu().numpy())

# Get predictions for Model #2
y_scores_model2 = []
with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model2(X_batch)
        y_scores_model2.extend(y_pred.cpu().numpy())

# Convert true labels to NumPy
y_true = y_test.cpu().numpy().flatten()
y_scores_model1 = np.array(y_scores_model1).flatten()
y_scores_model2 = np.array(y_scores_model2).flatten()

# Compute ROC curve and AUC for Model #1
fpr1, tpr1, _ = roc_curve(y_true, y_scores_model1)
roc_auc1 = auc(fpr1, tpr1)

# Compute ROC curve and AUC for Model #2
fpr2, tpr2, _ = roc_curve(y_true, y_scores_model2)
roc_auc2 = auc(fpr2, tpr2)

# Plot Combined ROC Curve
plt.figure(figsize=(8, 5))
plt.plot(fpr1, tpr1, color='blue', lw=2, label=f'Model #1 (AUC =
{roc_auc1:.4f})')
plt.plot(fpr2, tpr2, color='red', lw=2, linestyle="dashed",
label=f'Model #2 (AUC = {roc_auc2:.4f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle="dashed",
label="Random Guessing")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison: Model #1 vs. Model #2")
plt.legend(loc="lower right")
plt.show()
```



Combined Calibration Plot --

```
# Ensure both models are in evaluation mode
model.eval()
model2.eval()

# Get predictions for Model #1
y_scores_model1 = []
with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model(X_batch)
        y_scores_model1.extend(y_pred.cpu().numpy())

# Get predictions for Model #2
y_scores_model2 = []
with torch.no_grad():
    for X_batch, _ in test_loader:
        y_pred = model2(X_batch)
        y_scores_model2.extend(y_pred.cpu().numpy())

# Convert true labels to NumPy
y_true = y_test.cpu().numpy().flatten()
y_scores_model1 = np.array(y_scores_model1).flatten()
y_scores_model2 = np.array(y_scores_model2).flatten()

# Compute Calibration Curves
```

```
fraction_of_positives1, mean_predicted_value1 =
calibration_curve(y_true, y_scores_model1, n_bins=10)
fraction_of_positives2, mean_predicted_value2 =
calibration_curve(y_true, y_scores_model2, n_bins=10)

# Plot Combined Calibration Curve
plt.figure(figsize=(8, 5))
plt.plot(mean_predicted_value1, fraction_of_positives1,
marker="o", color='blue', label="Model #1")
plt.plot(mean_predicted_value2, fraction_of_positives2,
marker="s", color='red', linestyle="dashed", label="Model #2")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray",
label="Perfect Calibration")
plt.xlabel("Mean Predicted Probability")
plt.ylabel("Fraction of Positives")
plt.title("Calibration Plot - Model #1 vs. Model #2")
plt.legend()
plt.show()
```