Final Project League of Legends Match Predictor

June 28, 2025

1 Final Project: League of Legends Match Predictor

1.0.1 Introduction

League of Legends, a popular multiplayer online battle arena (MOBA) game, generates extensive data from matches, providing an excellent opportunity to apply machine learning techniques to real-world scenarios. Perform the following steps to build a logistic regression model aimed at predicting the outcomes of League of Legends matches.

Use the league_of_legends_data_large.csv file to perform the tasks.

1.0.2 Step 1: Data Loading and Preprocessing

Task 1: Load the League of Legends dataset and preprocess it for training. Loading and preprocessing the dataset involves reading the data, splitting it into training and testing sets, and standardizing the features. You will utilize `pandas` for data manipulation, `train_test_split` from `sklearn` for data splitting, and `StandardScaler` for feature scaling.

Note: Please ensure all the required libraries are installed and imported.

1 .Load the dataset: Use `pd.read_csv()` to load the dataset into a pandas DataFrame. 2. Split data into features and target: Separate win (target) and the remaining columns (features). X = data.drop('win', axis=1) y = data['win'] 3 .Split the Data into Training and Testing Sets: Use `train_test_split()` from `sklearn.model_selection` to divide the data. Set `test_size`=0.2 to allocate 20% for testing and 80% for training, and use `random_state`=42 to ensure reproducibility of the split. 4. Standardize the features: Use `StandardScaler()` from sklearn.preprocessing to scale the features. 5. Convert to PyTorch tensors: Use `torch.tensor()` to convert the data to PyTorch tensors.

Exercise 1: Write a code to load the dataset, split it into training and testing sets, standardize the features, and convert the data into PyTorch tensors for use in training a PyTorch model.

1.0.3 Setup

Installing required libraries:

The following required libraries are not pre-installed in the Skills Network Labs environment. You will need to run the following cell to install them:

[1]:
```
!pip install pandas
!pip install scikit-learn
```

```
!pip install torch
!pip install matplotlib
```

```
Collecting pandas
  Downloading
pandas-2.3.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata
(91 kB)
Collecting numpy>=1.26.0 (from pandas)
  Downloading numpy-2.3.1-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (62 kB)
Requirement already satisfied: python-dateutil>=2.8.2 in
/opt/conda/lib/python3.12/site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/conda/lib/python3.12/site-
packages (from pandas) (2024.2)
Collecting tzdata>=2022.7 (from pandas)
  Downloading tzdata-2025.2-py2.py3-none-any.whl.metadata (1.4 kB)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.12/site-
packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
Downloading
pandas-2.3.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.0
MB)
                         12.0/12.0 MB
170.3 MB/s eta 0:00:00
Downloading numpy-2.3.1-cp312-cp312-manylinux_2_28_x86_64.whl (16.6 MB)
                         16.6/16.6 MB
195.9 MB/s eta 0:00:00
Downloading tzdata-2025.2-py2.py3-none-any.whl (347 kB)
Installing collected packages: tzdata, numpy, pandas
Successfully installed numpy-2.3.1 pandas-2.3.0 tzdata-2025.2
Collecting scikit-learn
  Downloading scikit_learn-1.7.0-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (17 kB)
Requirement already satisfied: numpy>=1.22.0 in /opt/conda/lib/python3.12/site-
packages (from scikit-learn) (2.3.1)
Collecting scipy>=1.8.0 (from scikit-learn)
  Downloading
scipy-1.16.0-cp312-cp312-manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata
(61 kB)
Collecting joblib>=1.2.0 (from scikit-learn)
  Downloading joblib-1.5.1-py3-none-any.whl.metadata (5.6 kB)
Collecting threadpoolctl>=3.1.0 (from scikit-learn)
  Downloading threadpoolctl-3.6.0-py3-none-any.whl.metadata (13 kB)
Downloading
scikit_learn-1.7.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(12.5 MB)
                         12.5/12.5 MB
126.7 MB/s eta 0:00:00
Downloading joblib-1.5.1-py3-none-any.whl (307 kB)
Downloading
```

```
scipy-1.16.0-cp312-cp312-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (35.1
MB)
                         35.1/35.1 MB
202.6 MB/s eta 0:00:00
Downloading threadpoolctl-3.6.0-py3-none-any.whl (18 kB)
Installing collected packages: threadpoolctl, scipy, joblib, scikit-learn
Successfully installed joblib-1.5.1 scikit-learn-1.7.0 scipy-1.16.0
threadpoolctl-3.6.0
Collecting torch
  Downloading torch-2.7.1-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (29 kB)
Collecting filelock (from torch)
  Downloading filelock-3.18.0-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: typing-extensions>=4.10.0 in
/opt/conda/lib/python3.12/site-packages (from torch) (4.12.2)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.12/site-
packages (from torch) (75.8.0)
Collecting sympy>=1.13.3 (from torch)
  Downloading sympy-1.14.0-py3-none-any.whl.metadata (12 kB)
Collecting networkx (from torch)
  Downloading networkx-3.5-py3-none-any.whl.metadata (6.3 kB)
Requirement already satisfied: jinja2 in /opt/conda/lib/python3.12/site-packages
(from torch) (3.1.5)
Collecting fsspec (from torch)
  Downloading fsspec-2025.5.1-py3-none-any.whl.metadata (11 kB)
Collecting nvidia-cuda-nvrtc-cu12==12.6.77 (from torch)
  Downloading nvidia_cuda_nvrtc_cu12-12.6.77-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.6.77 (from torch)
  Downloading nvidia_cuda_runtime_cu12-12.6.77-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.6.80 (from torch)
  Downloading nvidia_cuda_cupti_cu12-12.6.80-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.5.1.17 (from torch)
  Downloading nvidia_cudnn_cu12-9.5.1.17-py3-none-
manylinux_2_28_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.6.4.1 (from torch)
  Downloading nvidia_cublas_cu12-12.6.4.1-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.3.0.4 (from torch)
  Downloading nvidia_cufft_cu12-11.3.0.4-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.7.77 (from torch)
  Downloading nvidia_curand_cu12-10.3.7.77-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.7.1.2 (from torch)
  Downloading nvidia_cusolver_cu12-11.7.1.2-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.6 kB)
```

```
Collecting nvidia-cusparse-cu12==12.5.4.2 (from torch)
  Downloading nvidia_cusparse_cu12-12.5.4.2-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparselt-cu12==0.6.3 (from torch)
  Downloading nvidia_cusparselt_cu12-0.6.3-py3-none-
manylinux2014_x86_64.whl.metadata (6.8 kB)
Collecting nvidia-nccl-cu12==2.26.2 (from torch)
  Downloading nvidia_nccl_cu12-2.26.2-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (2.0 kB)
Collecting nvidia-nvtx-cu12==12.6.77 (from torch)
  Downloading nvidia_nvtx_cu12-12.6.77-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-nvjitlink-cu12==12.6.85 (from torch)
  Downloading nvidia_nvjitlink_cu12-12.6.85-py3-none-
manylinux2010_x86_64.manylinux_2_12_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufile-cu12==1.11.1.6 (from torch)
  Downloading nvidia_cufile_cu12-1.11.1.6-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (1.5 kB)
Collecting triton==3.3.1 (from torch)
  Downloading triton-3.3.1-cp312-cp312-
manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl.metadata (1.5 kB)
Collecting mpmath<1.4,>=1.1.0 (from sympy>=1.13.3->torch)
  Downloading mpmath-1.3.0-py3-none-any.whl.metadata (8.6 kB)
Requirement already satisfied: MarkupSafe>=2.0 in
/opt/conda/lib/python3.12/site-packages (from jinja2->torch) (3.0.2)
Downloading torch-2.7.1-cp312-cp312-manylinux_2_28_x86_64.whl (821.0 MB)
                         821.0/821.0 MB
? eta 0:00:00 0:00:0100:02
Downloading nvidia_cublas_cu12-12.6.4.1-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (393.1 MB)
                         393.1/393.1 MB
17.3 MB/s eta 0:00:0000:0100:01
Downloading nvidia_cuda_cupti_cu12-12.6.80-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (8.9 MB)
                         8.9/8.9 MB
71.7 MB/s eta 0:00:00
Downloading nvidia_cuda_nvrtc_cu12-12.6.77-py3-none-
manylinux2014_x86_64.whl (23.7 MB)
                         23.7/23.7 MB
46.8 MB/s eta 0:00:00:00:01
Downloading nvidia_cuda_runtime_cu12-12.6.77-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (897 kB)
                         897.7/897.7 kB
58.1 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.5.1.17-py3-none-manylinux_2_28_x86_64.whl
(571.0 MB)
                         571.0/571.0 MB
4.1 MB/s eta 0:00:00:00:0100:01
```

```
Downloading nvidia_cufft_cu12-11.3.0.4-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (200.2 MB)
                         200.2/200.2 MB
27.4 MB/s eta 0:00:0000:0100:01
Downloading nvidia_cufile_cu12-1.11.1.6-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (1.1 MB)
                         1.1/1.1 MB
35.2 MB/s eta 0:00:00
Downloading nvidia_curand_cu12-10.3.7.77-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (56.3 MB)
                         56.3/56.3 MB
48.4 MB/s eta 0:00:00:00:01
Downloading nvidia_cusolver_cu12-11.7.1.2-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (158.2 MB)
                         158.2/158.2 MB
48.7 MB/s eta 0:00:0000:0100:01
Downloading nvidia_cusparse_cu12-12.5.4.2-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (216.6 MB)
                         216.6/216.6 MB
21.3 MB/s eta 0:00:0000:0100:01
Downloading nvidia_cusparselt_cu12-0.6.3-py3-none-manylinux2014_x86_64.whl
(156.8 MB)
                         156.8/156.8 MB
48.8 MB/s eta 0:00:0000:0100:01
Downloading nvidia_nccl_cu12-2.26.2-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (201.3 MB)
                         201.3/201.3 MB
14.9 MB/s eta 0:00:0000:0100:01
Downloading nvidia_nvjitlink_cu12-12.6.85-py3-none-
manylinux2010_x86_64.manylinux_2_12_x86_64.whl (19.7 MB)
                         19.7/19.7 MB
48.8 MB/s eta 0:00:00:00:01
Downloading nvidia_nvtx_cu12-12.6.77-py3-none-
manylinux2014_x86_64.manylinux_2_17_x86_64.whl (89 kB)
Downloading
triton-3.3.1-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl (155.7
MB)
                         155.7/155.7 MB
32.8 MB/s eta 0:00:0000:0100:01
Downloading sympy-1.14.0-py3-none-any.whl (6.3 MB)
                         6.3/6.3 MB
1.2 MB/s eta 0:00:000:00:01
Downloading filelock-3.18.0-py3-none-any.whl (16 kB)
Downloading fsspec-2025.5.1-py3-none-any.whl (199 kB)
Downloading networkx-3.5-py3-none-any.whl (2.0 MB)
                         2.0/2.0 MB
46.5 MB/s eta 0:00:00
Downloading mpmath-1.3.0-py3-none-any.whl (536 kB)
```

```
                         536.2/536.2 kB
30.1 MB/s eta 0:00:00
Installing collected packages: nvidia-cusparselt-cu12, mpmath, triton, sympy,
nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12,
nvidia-cufile-cu12, nvidia-cuda-runtime-cu12, nvidia-cuda-nvrtc-cu12, nvidia-
cuda-cupti-cu12, nvidia-cublas-cu12, networkx, fsspec, filelock, nvidia-
cusparse-cu12, nvidia-cufft-cu12, nvidia-cudnn-cu12, nvidia-cusolver-cu12, torch
Successfully installed filelock-3.18.0 fsspec-2025.5.1 mpmath-1.3.0 networkx-3.5
nvidia-cublas-cu12-12.6.4.1 nvidia-cuda-cupti-cu12-12.6.80 nvidia-cuda-nvrtc-
cu12-12.6.77 nvidia-cuda-runtime-cu12-12.6.77 nvidia-cudnn-cu12-9.5.1.17 nvidia-
cufft-cu12-11.3.0.4 nvidia-cufile-cu12-1.11.1.6 nvidia-curand-cu12-10.3.7.77
nvidia-cusolver-cu12-11.7.1.2 nvidia-cusparse-cu12-12.5.4.2 nvidia-cusparselt-
cu12-0.6.3 nvidia-nccl-cu12-2.26.2 nvidia-nvjitlink-cu12-12.6.85 nvidia-nvtx-
cu12-12.6.77 sympy-1.14.0 torch-2.7.1 triton-3.3.1
Collecting matplotlib
  Downloading matplotlib-3.10.3-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (11 kB)
Collecting contourpy>=1.0.1 (from matplotlib)
  Downloading contourpy-1.3.2-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (5.5 kB)
Collecting cycler>=0.10 (from matplotlib)
  Downloading cycler-0.12.1-py3-none-any.whl.metadata (3.8 kB)
Collecting fonttools>=4.22.0 (from matplotlib)
  Downloading fonttools-4.58.4-cp312-cp312-
manylinux1_x86_64.manylinux2014_x86_64.manylinux_2_17_x86_64.manylinux_2_5_x86_6
4.whl.metadata (106 kB)
Collecting kiwisolver>=1.3.1 (from matplotlib)
  Downloading kiwisolver-1.4.8-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (6.2 kB)
Requirement already satisfied: numpy>=1.23 in /opt/conda/lib/python3.12/site-
packages (from matplotlib) (2.3.1)
Requirement already satisfied: packaging>=20.0 in
/opt/conda/lib/python3.12/site-packages (from matplotlib) (24.2)
Collecting pillow>=8 (from matplotlib)
  Downloading pillow-11.2.1-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (8.9
kB)
Collecting pyparsing>=2.3.1 (from matplotlib)
  Downloading pyparsing-3.2.3-py3-none-any.whl.metadata (5.0 kB)
Requirement already satisfied: python-dateutil>=2.7 in
/opt/conda/lib/python3.12/site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.12/site-
packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
Downloading
matplotlib-3.10.3-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(8.6 MB)
                         8.6/8.6 MB
124.5 MB/s eta 0:00:00
Downloading
```

```
contourpy-1.3.2-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (323
kB)
Downloading cycler-0.12.1-py3-none-any.whl (8.3 kB)
Downloading fonttools-4.58.4-cp312-cp312-
manylinux1_x86_64.manylinux2014_x86_64.manylinux_2_17_x86_64.manylinux_2_5_x86_6
4.whl (4.9 MB)
                         4.9/4.9 MB
144.3 MB/s eta 0:00:00
Downloading
kiwisolver-1.4.8-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.5
MB)
                         1.5/1.5 MB
93.4 MB/s eta 0:00:00
Downloading pillow-11.2.1-cp312-cp312-manylinux_2_28_x86_64.whl (4.6 MB)
                         4.6/4.6 MB
152.5 MB/s eta 0:00:00
Downloading pyparsing-3.2.3-py3-none-any.whl (111 kB)
Installing collected packages: pyparsing, pillow, kiwisolver, fonttools, cycler,
contourpy, matplotlib
Successfully installed contourpy-1.3.2 cycler-0.12.1 fonttools-4.58.4
kiwisolver-1.4.8 matplotlib-3.10.3 pillow-11.2.1 pyparsing-3.2.3
```

[3]:
```python
# Task 1: Data Loading and Preprocessing

# Install necessary libraries if running in a new environment
# !pip install pandas scikit-learn torch matplotlib

import pandas as pd
import numpy as np
import torch
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 1. Load the dataset
df = pd.read_csv('league_of_legends_data_large.csv')

# 2. Split data into features (X) and target (y)
X = df.drop('win', axis=1)
y = df['win']

# 3. Split into training and testing sets (80/20 split)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 4. Standardize the features
scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 5. Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32).view(-1, 1)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32).view(-1, 1)

# Checking the Output shapes
print("X_train shape:", X_train_tensor.shape)
print("y_train shape:", y_train_tensor.shape)
```

```
X_train shape: torch.Size([800, 8])
y_train shape: torch.Size([800, 1])
```

1.0.4 Step 2: Logistic Regression Model

Task 2: Implement a logistic regression model using PyTorch. Defining the logistic regression model involves specifying the input dimensions, the forward pass using the sigmoid activation function, and initializing the model, loss function, and optimizer.

1 .Define the Logistic Regression Model: Create a class LogisticRegressionModel that inherits from torch.nn.Module. - In the `__init__()` method, define a linear layer (nn.Linear) to implement the logistic regression model. - The `forward()` method should apply the sigmoid activation function to the output of the linear layer.

2.Initialize the Model, Loss Function, and Optimizer: - Set input_dim: Use `X_train.shape[1]` to get the number of features from the training data (X_train). - Initialize the model: Create an instance of the LogisticRegressionModel class (e.g., `model = LogisticRegressionModel()`)while passing input_dim as a parameter - Loss Function: Use `BCELoss()` from torch.nn (Binary Cross-Entropy Loss). - Optimizer: Initialize the optimizer using `optim.SGD()` with a learning rate of 0.01

Exercise 2: Define the logistic regression model using PyTorch, specifying the input dimensions and the forward pass. Initialize the model, loss function, and optimizer.

[4]:
```
import torch.nn as nn
import torch.optim as optim

# 1. Define the Logistic Regression Model
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_dim):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(input_dim, 1)  # One output for binary␣
 ↪classification

    def forward(self, x):
        return torch.sigmoid(self.linear(x))  # Apply sigmoid to get probability
```

```python
# 2. Initialize model, loss function, and optimizer
input_dim = X_train_tensor.shape[1]  # Number of features
model = LogisticRegressionModel(input_dim)

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

# SGD Optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Check model architecture
print(model)
```

```
LogisticRegressionModel(
  (linear): Linear(in_features=8, out_features=1, bias=True)
)
```

1.0.5 Step 3: Model Training

Task 3: Train the logistic regression model on the dataset. The training loop will run for a specified number of epochs. In each epoch, the model makes predictions, calculates the loss, performs backpropagation, and updates the model parameters.

1. Set Number of Epochs:
    - Define the number of epochs for training to 1000.
2. Training Loop:
   For each epoch:
    - Set the model to training mode using `model.train()`.
    - Zero the gradients using `optimizer.zero_grad()`.
    - Pass the training data (`X_train`) through the model to get the predictions (`outputs`).
    - Calculate the loss using the defined loss function (`criterion`).
    - Perform backpropagation with `loss.backward()`.
    - Update the model's weights using `optimizer.step()`.
3. Print Loss Every 100 Epochs:
    - After every 100 epochs, print the current epoch number and the loss value.
4. Model Evaluation:
    - Set the model to evaluation mode using `model.eval()`.
    - Use `torch.no_grad()` to ensure no gradients are calculated during evaluation.
    - Get predictions on both the training set (`X_train`) and the test set (`X_test`).
5. Calculate Accuracy:
    - For both the training and test datasets, compute the accuracy by comparing the predicted values with the true values (`y_train`, `y_test`).
    - Use a threshold of 0.5 for classification
6. Print Accuracy:
    - Print the training and test accuracies after the evaluation is complete.

Exercise 3: Write the code to train the logistic regression model on the dataset. Implement the training loop, making predictions, calculating the loss, performing backpropagation, and updating model parameters. Evaluate the model's accuracy on training and testing sets.

[5]:
```python
# Task 3: Train the logistic regression model

# Number of epochs
epochs = 1000

for epoch in range(epochs):
    # Set model to training mode
    model.train()

    # Forward pass
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Print loss every 100 epochs
    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}")

# Evaluate the model
model.eval()
with torch.no_grad():
    train_preds = model(X_train_tensor)
    test_preds = model(X_test_tensor)

# Apply threshold to get class labels (0 or 1)
train_preds_class = (train_preds >= 0.5).float()
test_preds_class = (test_preds >= 0.5).float()

# Calculate accuracy
train_accuracy = (train_preds_class == y_train_tensor).float().mean().item()
test_accuracy = (test_preds_class == y_test_tensor).float().mean().item()

print(f"\n Training Accuracy: {train_accuracy * 100:.2f}%")
print(f" Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
Epoch [100/1000], Loss: 0.7220
Epoch [200/1000], Loss: 0.7076
Epoch [300/1000], Loss: 0.6988
Epoch [400/1000], Loss: 0.6935
Epoch [500/1000], Loss: 0.6902
Epoch [600/1000], Loss: 0.6883
Epoch [700/1000], Loss: 0.6871
Epoch [800/1000], Loss: 0.6864
Epoch [900/1000], Loss: 0.6859
Epoch [1000/1000], Loss: 0.6857

 Training Accuracy: 53.87%
 Test Accuracy: 51.00%
```

1.0.6 Step 4: Model Optimization and Evaluation

Task 4: Implement optimization techniques and evaluate the model's performance. Optimization techniques such as L2 regularization (Ridge Regression) help in preventing overfitting. The model is retrained with these optimizations, and its performance is evaluated on both training and testing sets.

Weight Decay :In the context of machine learning and specifically in optimization algorithms, weight_decay is a parameter used to apply L2 regularization to the model's parameters (weights). It helps prevent the model from overfitting by penalizing large weight values, thereby encouraging the model to find simpler solutions.To use L2 regularization, you need to modify the optimizer by setting the weight_decay parameter. The weight_decay parameter in the optimizer adds the L2 regularization term during training. For example, when you initialize the optimizer with optim.SGD(model.parameters(), lr=0.01, weight_decay=0.01), the weight_decay=0.01 term applies L2 regularization with a strength of 0.01.

1. Set Up the Optimizer with L2 Regularization:
   - Modify the optimizer to include `weight_decay` for L2 regularization.
   - Example:
     `optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=0.01)`
2. Train the Model with L2 Regularization:
   - Follow the same steps as before but use the updated optimizer with regularization during training.
   - Use epochs=1000
3. Evaluate the Optimized Model:
   - After training, evaluate the model on both the training and test datasets.
   - Compute the accuracy for both sets by comparing the model's predictions to the true labels (`y_train` and `y_test`).
4. Calculate and Print the Accuracy:
   - Use a threshold of 0.5 to determine whether the model's predictions are class 0 or class 1.
   - Print the training accuracy and test accuracy after evaluation.

Exercise 4: Implement optimization techniques like L2 regularization and retrain the model. Evaluate the performance of the optimized model on both training and testing sets.

[6]:
```python
# Task 4: Optimization with L2 Regularization

# Re-initialize the model
model_l2 = LogisticRegressionModel(input_dim)
```

```python
# Define optimizer with L2 regularization (weight decay)
optimizer_l2 = optim.SGD(model_l2.parameters(), lr=0.01, weight_decay=0.01)

# Loss function remains the same
criterion_l2 = nn.BCELoss()

# Number of epochs
epochs = 1000

# Training loop with L2 regularization
for epoch in range(epochs):
    model_l2.train()

    # Forward pass
    outputs = model_l2(X_train_tensor)
    loss = criterion_l2(outputs, y_train_tensor)

    # Backward pass and optimization
    optimizer_l2.zero_grad()
    loss.backward()
    optimizer_l2.step()

    if (epoch + 1) % 100 == 0:
        print(f"[L2] Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}")

# Evaluate the L2-optimized model
model_l2.eval()
with torch.no_grad():
    train_preds_l2 = model_l2(X_train_tensor)
    test_preds_l2 = model_l2(X_test_tensor)

# Convert probabilities to binary predictions
train_preds_l2_class = (train_preds_l2 >= 0.5).float()
test_preds_l2_class = (test_preds_l2 >= 0.5).float()

# Compute accuracy
train_acc_l2 = (train_preds_l2_class == y_train_tensor).float().mean().item()
test_acc_l2 = (test_preds_l2_class == y_test_tensor).float().mean().item()

print(f"\n  [L2] Training Accuracy: {train_acc_l2 * 100:.2f}%")
print(f"  [L2] Test Accuracy: {test_acc_l2 * 100:.2f}%")
```

```
[L2] Epoch [100/1000], Loss: 0.7170
[L2] Epoch [200/1000], Loss: 0.7049
[L2] Epoch [300/1000], Loss: 0.6973
[L2] Epoch [400/1000], Loss: 0.6925
[L2] Epoch [500/1000], Loss: 0.6896
```

```
[L2] Epoch [600/1000], Loss: 0.6879
[L2] Epoch [700/1000], Loss: 0.6868
[L2] Epoch [800/1000], Loss: 0.6862
[L2] Epoch [900/1000], Loss: 0.6858
[L2] Epoch [1000/1000], Loss: 0.6856

 [L2] Training Accuracy: 55.75%
 [L2] Test Accuracy: 51.00%
```

1.0.7 Step 5: Visualization and Interpretation

Visualization tools like confusion matrices and ROC curves provide insights into the model's performance. The confusion matrix helps in understanding the classification accuracy, while the ROC curve illustrates the trade-off between sensitivity and specificity.

Confusion Matrix : A Confusion Matrix is a fundamental tool used in classification problems to evaluate the performance of a model. It provides a matrix showing the number of correct and incorrect predictions made by the model, categorized by the actual and predicted classes. Where - True Positive (TP): Correctly predicted positive class (class 1). - True Negative (TN): Correctly predicted negative class (class 0). - False Positive (FP): Incorrectly predicted as positive (class 1), but the actual class is negative (class 0). This is also called a Type I error. - False Negative (FN): Incorrectly predicted as negative (class 0), but the actual class is positive (class 1). This is also called a Type II error.

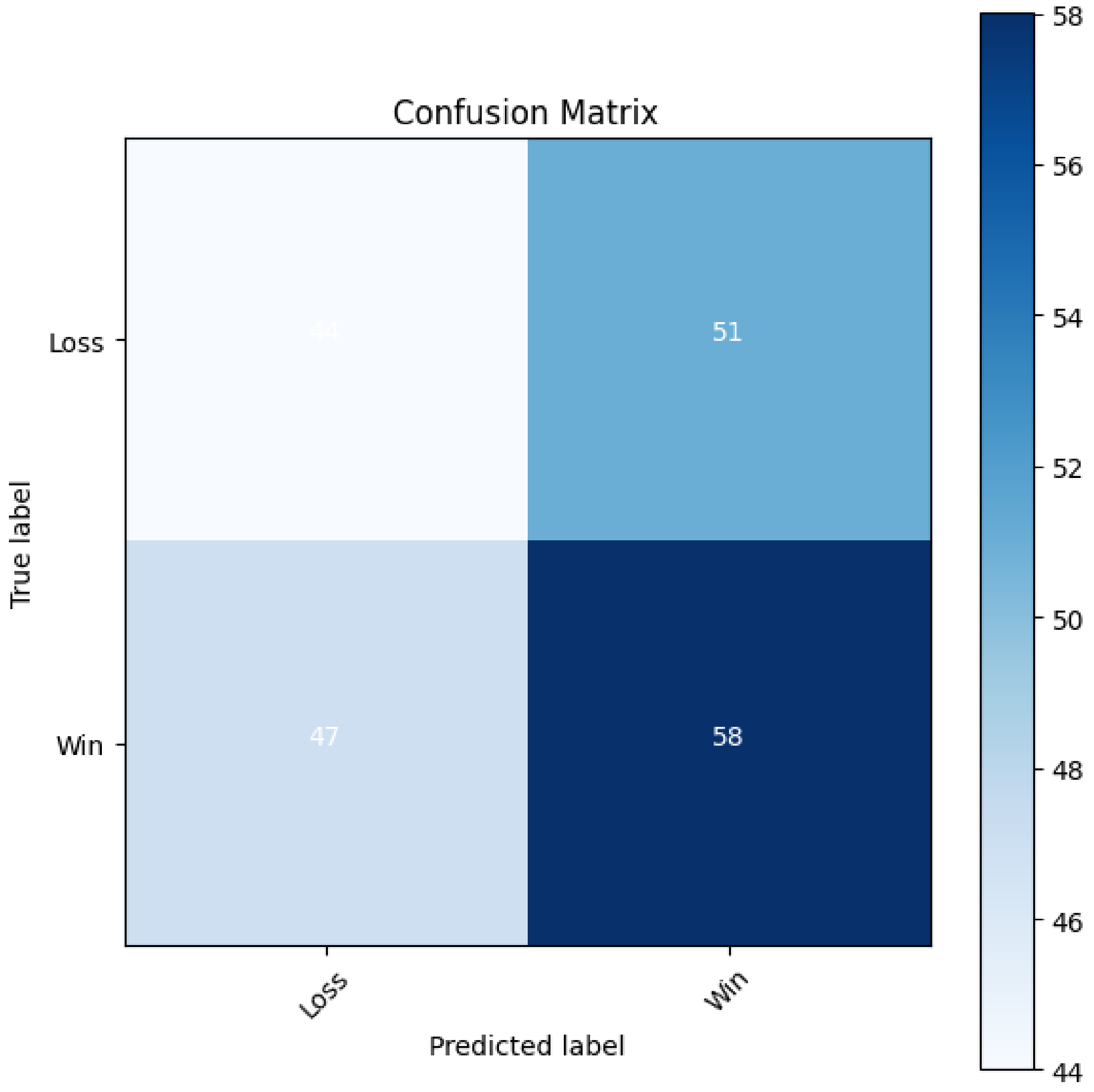
ROC Curve (Receiver Operating Characteristic Curve): The ROC Curve is a graphical representation used to evaluate the performance of a binary classification model across all classification thresholds. It plots two metrics: - True Positive Rate (TPR) or Recall (Sensitivity)-It is the proportion of actual positive instances (class 1) that were correctly classified as positive by the model. - False Positive Rate (FPR)-It is the proportion of actual negative instances (class 0) that were incorrectly classified as positive by the model.

AUC: AUC stands for Area Under the Curve and is a performance metric used to evaluate the quality of a binary classification model. Specifically, it refers to the area under the ROC curve (Receiver Operating Characteristic curve), which plots the True Positive Rate (TPR) versus the False Positive Rate (FPR) for different threshold values.

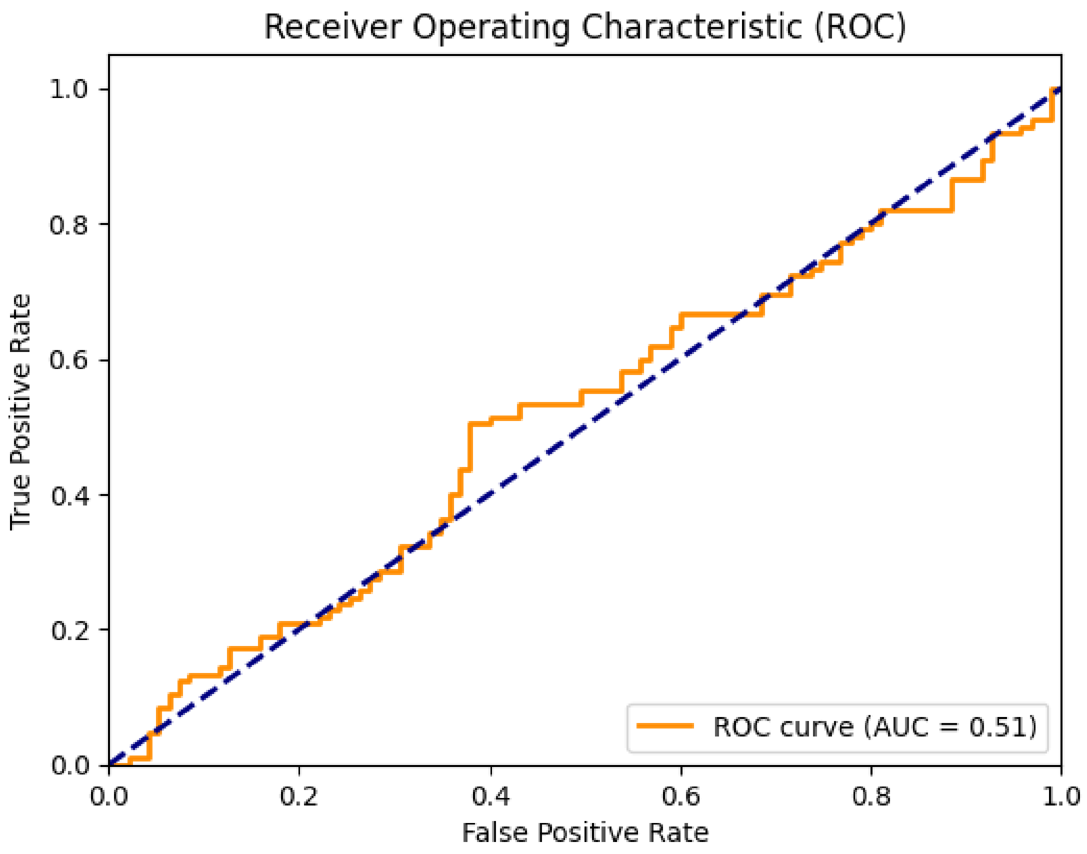
Classification Report: A Classification Report is a summary of various classification metrics, which are useful for evaluating the performance of a classifier on the given dataset.

Exercise 5: Write code to visualize the model's performance using confusion matrices and ROC curves. Generate classification reports to evaluate precision, recall, and F1-score. Retrain the model with L2 regularization and evaluate the performance.

[8]:
```python
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report, roc_curve,␣
 ↪auc
import itertools

# Task 5: Model Evaluation and Visualization

# Convert tensors back to numpy arrays for sklearn
y_test_np = y_test_tensor.numpy()
y_pred_test = test_preds_l2.numpy()
y_pred_test_labels = (y_pred_test >= 0.5).astype(int)

# === Confusion Matrix ===
cm = confusion_matrix(y_test_np, y_pred_test_labels)

plt.figure(figsize=(6, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = range(2)
plt.xticks(tick_marks, ['Loss', 'Win'], rotation=45)
plt.yticks(tick_marks, ['Loss', 'Win'])

# Add numbers inside the matrix
thresh = cm.max() / 2
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j],
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()

# === Classification Report ===
print("Classification Report:\n")
print(classification_report(y_test_np, y_pred_test_labels,␣
 ↪target_names=['Loss', 'Win']))

# === ROC Curve and AUC ===
fpr, tpr, thresholds = roc_curve(y_test_np, y_pred_test)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.
 ↪2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
plt.show()
```



```
Classification Report:
              precision    recall  f1-score   support

        Loss       0.48      0.46      0.47        95
         Win       0.53      0.55      0.54       105

    accuracy                           0.51       200
   macro avg       0.51      0.51      0.51       200
weighted avg       0.51      0.51      0.51       200
```



Double-click here for the Hint.

1.0.8 Step 6: Model Saving and Loading

Task 6: Save and load the trained model. This task demonstrates the techniques to persist a trained model using `torch.save` and reload it using `torch.load`. Evaluating the loaded model ensures that it retains its performance, making it practical for deployment in real-world applications.

1. Saving the Model:

- Save the model's learned weights and biases using torch.save().(e.g. , torch.save(model.state_dict(), 'your_model_name.pth'))
- Saving only the state dictionary (model parameters) is preferred because it's more flexible and efficient than saving the entire model object.

2. Loading the Model:

- Create a new model instance (e.g., `model = LogisticRegressionModel()`) and load the saved parameters. (e.g. , `model.load_state_dict(torch.load('your_model_name.pth'))`)'.

3. Evaluating the Loaded Model:
    - After loading, set the model to evaluation mode by calling 'model.eval()

- After loading the model, evaluate it again on the test dataset to make sure it performs similarly to when it was first trained..Now evaluate it on the test data.
- Use `torch.no_grad()` to ensure that no gradients are computed.

Exercise 6: Write code to save the trained model and reload it. Ensure the loaded model performs consistently by evaluating it on the test dataset.

[9]:
```
# Task 6: Save and Load the Model

# === 1. Save the model's state_dict ===
model_path = "logistic_model_l2.pth"
torch.save(model_l2.state_dict(), model_path)
print(f" Model saved to: {model_path}")

# === 2. Load the model's state_dict into a new model ===
# Re-create the model architecture
loaded_model = LogisticRegressionModel(input_dim)
loaded_model.load_state_dict(torch.load(model_path))

# === 3. Evaluate the loaded model ===
loaded_model.eval()
with torch.no_grad():
    loaded_preds = loaded_model(X_test_tensor)
    loaded_preds_class = (loaded_preds >= 0.5).float()

# Calculate accuracy
loaded_test_accuracy = (loaded_preds_class == y_test_tensor).float().mean().
 ↪item()
print(f"\n Loaded Model Test Accuracy: {loaded_test_accuracy * 100:.2f}%")
```

```
Model saved to: logistic_model_l2.pth

Loaded Model Test Accuracy: 51.00%
```

1.0.9 Step 7: Hyperparameter Tuning

Task 7: Perform hyperparameter tuning to find the best learning rate. By testing different learning rates, you will identify the optimal rate that provides the best test accuracy. This fine-tuning is crucial for enhancing model performance . 1. Define Learning Rates: - Choose these learning rates to test ,[0.01, 0.05, 0.1]

2. Reinitialize the Model for Each Learning Rate:

- For each learning rate, you'll need to reinitialize the model and optimizer e.g.(`torch.optim.SGD(model.parameters(), lr=lr)`).
- Each new learning rate requires reinitializing the model since the optimizer and its parameters are linked to the learning rate.

3. Train the Model for Each Learning Rate:

- Train the model for a fixed number of epochs (e.g., 50 or 100 epochs) for each learning rate, and compute the accuracy on the test set.
- Track the test accuracy for each learning rate and identify which one yields the best performance.

4. Evaluate and Compare:

- After training with each learning rate, compare the test accuracy for each configuration.
- Report the learning rate that gives the highest test accuracy

Exercise 7: Perform hyperparameter tuning to find the best learning rate. Retrain the model for each learning rate and evaluate its performance to identify the optimal rate.

[10]:
```
# Task 7: Hyperparameter Tuning

learning_rates = [0.01, 0.05, 0.1]
results = {}

for lr in learning_rates:
    print(f"\n  Training with learning rate: {lr}")

    # Reinitialize model and optimizer
    model_tune = LogisticRegressionModel(input_dim)
    optimizer_tune = optim.SGD(model_tune.parameters(), lr=lr, weight_decay=0.
 ↪01)
    criterion_tune = nn.BCELoss()

    # Train for 100 epochs
    epochs = 100
    for epoch in range(epochs):
        model_tune.train()
        outputs = model_tune(X_train_tensor)
        loss = criterion_tune(outputs, y_train_tensor)

        optimizer_tune.zero_grad()
        loss.backward()
        optimizer_tune.step()

    # Evaluate on test set
    model_tune.eval()
    with torch.no_grad():
        test_preds = model_tune(X_test_tensor)
        test_preds_class = (test_preds >= 0.5).float()
        test_accuracy = (test_preds_class == y_test_tensor).float().mean().
 ↪item()

    print(f"  Test Accuracy at LR={lr}: {test_accuracy * 100:.2f}%")
    results[lr] = test_accuracy
```

```
# Find the best learning rate
best_lr = max(results, key=results.get)
print(f"\n Best Learning Rate: {best_lr} with Test Accuracy: {results[best_lr]␣
↪* 100:.2f}%")
```

```
Training with learning rate: 0.01
Test Accuracy at LR=0.01: 47.50%

Training with learning rate: 0.05
Test Accuracy at LR=0.05: 48.50%

Training with learning rate: 0.1
Test Accuracy at LR=0.1: 51.50%

Best Learning Rate: 0.1 with Test Accuracy: 51.50%
```

1.0.10 Step 8: Feature Importance

Task 8: Evaluate feature importance to understand the impact of each feature on the prediction. The code to evaluate feature importance to understand the impact of each feature on the prediction.

1.Extracting Model Weights: - The weights of the logistic regression model represent the importance of each feature in making predictions. These weights are stored in the model's linear layer (`model.linear.weight`). - You can extract the weights using `model.linear.weight.data.numpy()` and flatten the resulting tensor to get a 1D array of feature importances.

2.Creating a DataFrame: - Create a pandas DataFrame with two columns: one for the feature names and the other for their corresponding importance values (i.e., the learned weights). - Ensure the features are aligned with their names in your dataset (e.g., 'X_train.columns).

3. Sorting and Plotting Feature Importance:

- Sort the features based on the absolute value of their importance (weights) to identify the most impactful features.
- Use a bar plot (via `matplotlib`) to visualize the sorted feature importances, with the feature names on the y-axis and importance values on the x-axis.

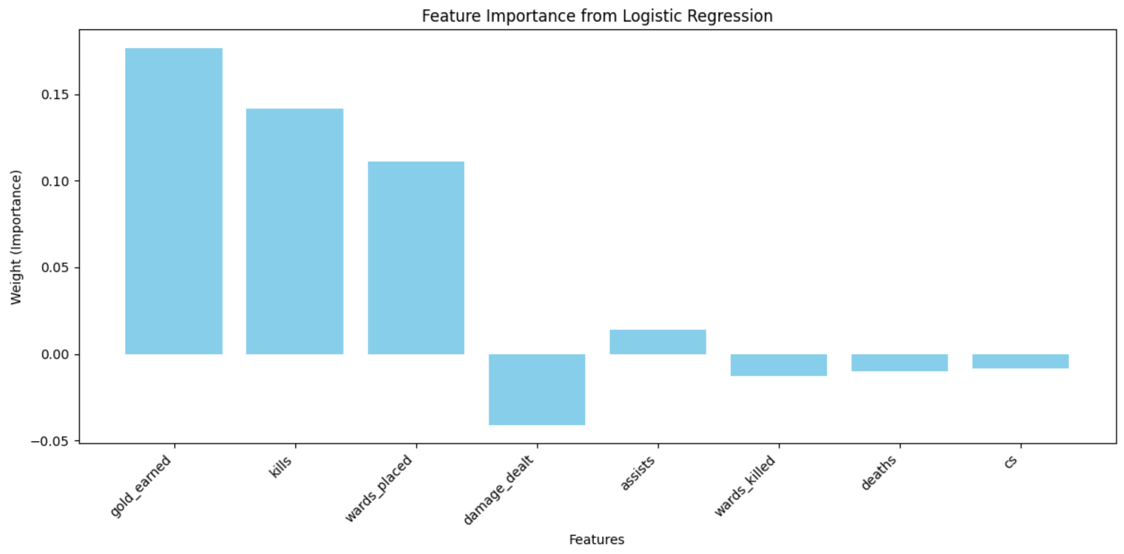
4. Interpreting the Results:

- Larger absolute weights indicate more influential features. Positive weights suggest a positive correlation with the outcome (likely to predict the positive class), while negative weights suggest the opposite.

Exercise 8: Evaluate feature importance by extracting the weights of the linear layer and creating a DataFrame to display the importance of each feature. Visualize the feature importance using a bar plot.

[11]:
```python
# Task 8: Feature Importance Analysis

import pandas as pd
import matplotlib.pyplot as plt

# Extract weights from the trained L2-regularized model
weights = model_l2.linear.weight.data.numpy().flatten()

# Get original feature names
features = X.columns

# Create a DataFrame for feature importance
feature_importance = pd.DataFrame({
    'Feature': features,
    'Importance': weights
})

# Sort by absolute importance
feature_importance['AbsImportance'] = feature_importance['Importance'].abs()
feature_importance = feature_importance.sort_values(by='AbsImportance',␣
 ↪ascending=False)

# Display the sorted DataFrame
print("  Feature Importance:\n")
print(feature_importance[['Feature', 'Importance']])

# Plot feature importance
plt.figure(figsize=(12, 6))
plt.bar(feature_importance['Feature'], feature_importance['Importance'],␣
 ↪color='skyblue')
plt.xlabel('Features')
plt.ylabel('Weight (Importance)')
plt.title('Feature Importance from Logistic Regression')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

```
  Feature Importance:

        Feature  Importance
3   gold_earned    0.176423
0         kills    0.141373
5  wards_placed    0.110920
7  damage_dealt   -0.040911
2       assists    0.013988
6   wards_killed  -0.012527
1        deaths   -0.010284
4            cs   -0.008212
```



Double-click here for the Hint

Conclusion: Congratulations on completing the project! In this final project, you built a logistic regression model to predict the outcomes of League of Legends matches based on various in-game statistics. This comprehensive project involved several key steps, including data loading and preprocessing, model implementation, training, optimization, evaluation, visualization, model saving and loading, hyperparameter tuning, and feature importance analysis. This project provided hands-on experience with the complete workflow of developing a machine learning model for binary classification tasks using PyTorch.

```
[ ]:
```