Final Project League of Legends Match Predictor-v2

January 23, 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
```

```
Requirement already satisfied: pandas in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (2.2.2)
Requirement already satisfied: numpy>=1.23.2 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
pandas) (2024.2)
Requirement already satisfied: six>=1.5 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from python-dateutil>=2.8.2->pandas)
(1.17.0)
Requirement already satisfied: scikit-learn in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (1.5.0)
Requirement already satisfied: numpy>=1.19.5 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from scikit-learn) (1.15.1)
Requirement already satisfied: joblib>=1.2.0 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
scikit-learn) (3.5.0)
Requirement already satisfied: torch in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (2.3.1)
Requirement already satisfied: filelock in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from torch) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
torch) (4.12.2)
Requirement already satisfied: sympy in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from torch) (1.13.1)
Requirement already satisfied: networkx in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from torch) (3.1.5)
Requirement already satisfied: fsspec in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from torch) (2024.12.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
jinja2->torch) (3.0.2)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
```

```
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
sympy->torch) (1.3.0)
Requirement already satisfied: matplotlib in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (3.8.0)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (4.55.3)
Requirement already satisfied: kiwisolver>=1.0.1 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (1.4.8)
Requirement already satisfied: numpy<2,>=1.21 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (24.2)
Requirement already satisfied: pillow>=6.2.0 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from matplotlib) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in
/Users/kevin/anaconda3/envs/mps-python-3.11/lib/python3.11/site-packages (from
matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /Users/kevin/anaconda3/envs/mps-
python-3.11/lib/python3.11/site-packages (from python-dateutil>=2.7->matplotlib)
(1.17.0)
```

[11]:
```python
## Write your code here
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch


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

display(data)

# 2. Split data into features and target:
# Separate win (target) and the remaining columns (features).</br>
```

```
X = data.drop('win', axis=1)
y = data['win']

# 3 .Split the Data into Training and Testing Sets:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42, stratify=y)

display(f'X_train shape: {X_train.shape}')
display(f'y_train shape: {y_train.shape}')
display(f'X_test shape: {X_test.shape}')
display(f'y_test shape: {y_test.shape}')
# 4. Standardize the features:
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 5. Convert to PyTorch tensors:
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train.values, dtype=torch.float32)
y_test = torch.tensor(y_test.values, dtype=torch.float32)
```

```
     win  kills  deaths  assists  gold_earned   cs  wards_placed  \
0      0     16       6       19        17088  231            11
1      1      8       8        5        14865  259            10
2      0      0      17       11        15919  169            14
3      0     19      11        1        11534  264            14
4      0     12       7        6        18926  124            15
..   …     …      …       …          …   …            …
995    0      2      15       12        17170  294             8
996    0      5      13        4        19524  236            14
997    1      8       7        8         7961  139            11
998    1      5      17        5         8226  193             9
999    0      8       9        4        18878   55            18

     wards_killed  damage_dealt
0               7         15367
1               2         38332
2               5         24642
3               3         15789
4               7         40268
..            …           …
995             6         33469
996             3          8845
997             7         49650
998             9         28290
999             9         35699
```

```
[1000 rows x 9 columns]

'X_train shape: (800, 8)'

'y_train shape: (800,)'

'X_test shape: (200, 8)'

'y_test shape: (200,)'
```

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.

[12]:
```python
## Write your code here
import torch.nn as nn
import torch.optim as optim

# 1 .Define the Logistic Regression Model:
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_units=30, output_units=1):
        super(LogisticRegressionModel, self).__init__()
        self.fc1 = nn.Linear(input_units, output_units)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        return x


# 2.Initialize the Model, Loss Function, and Optimizer:
model = LogisticRegressionModel(input_units=X_train.shape[1], output_units=1)
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

[13]:
```
display(model)
```

```
LogisticRegressionModel(
  (fc1): 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.

[14]:
```
# Write your code here
# 1. Set Number of Epochs:
epochs = 1000

for epoch in range(epochs):
   # 2. Training Loop:
   model.train()
   optimizer.zero_grad()
   outputs = model(X_train)
```

```
    train_loss = criterion(outputs, y_train.view(-1, 1))
    train_loss.backward()
    optimizer.step()
    # 3. Print Loss Every 100 Epochs:
    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch + 1}/{epochs}], Train Loss: {train_loss:.4f}')

# 4. Model Evaluation:
model.eval()
with torch.no_grad():
    train_preds = model(X_train)
    test_preds = model(X_test)

train_preds = (train_preds >= 0.5).float()
test_preds = (test_preds >= 0.5).float()


# 5. Calculate accuracy
train_accuracy = (train_preds.eq(y_train.view(-1, 1)).sum().item()) / y_train.
 ↪shape[0]
test_accuracy = (test_preds.eq(y_test.view(-1, 1)).sum().item()) / y_test.
 ↪shape[0]

# 6. Print final accuracy
print(f'Training Accuracy: {train_accuracy * 100:.2f}%')
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

```
Training Accuracy: 53.50%
Test Accuracy: 53.50%
```

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.

[15]:
```
## Write your code here
# 1. Set Up the Optimizer with L2 Regularization:
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=0.01)

# 1. Set Number of Epochs:
epochs = 1000

for epoch in range(epochs):
   # 2. Training Loop:
   model.train()
   optimizer.zero_grad()
   outputs = model(X_train)
   train_loss = criterion(outputs, y_train.view(-1, 1))
   train_loss.backward()
   optimizer.step()
   # 3. Print Loss Every 100 Epochs:
   if (epoch + 1) % 100 == 0:
      print(f'Epoch [{epoch + 1}/{epochs}], Train Loss: {train_loss:.4f}')

# 4. Model Evaluation:
model.eval()
with torch.no_grad():
   train_preds = model(X_train)
   test_preds = model(X_test)

train_preds = (train_preds >= 0.5).float()
test_preds = (test_preds >= 0.5).float()


# 5. Calculate accuracy
```

```
train_accuracy = (train_preds.eq(y_train.view(-1, 1)).sum().item()) / y_train.
↪shape[0]
test_accuracy = (test_preds.eq(y_test.view(-1, 1)).sum().item()) / y_test.
↪shape[0]

# 6. Print final accuracy
print(f'Training Accuracy: {train_accuracy * 100:.2f}%')
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

```
Epoch [100/1000], Train Loss: 0.6907
Epoch [200/1000], Train Loss: 0.6906
Epoch [300/1000], Train Loss: 0.6905
Epoch [400/1000], Train Loss: 0.6904
Epoch [500/1000], Train Loss: 0.6904
Epoch [600/1000], Train Loss: 0.6903
Epoch [700/1000], Train Loss: 0.6903
Epoch [800/1000], Train Loss: 0.6903
Epoch [900/1000], Train Loss: 0.6903
Epoch [1000/1000], Train Loss: 0.6903
Training Accuracy: 52.25%
Test Accuracy: 54.00%
```

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.

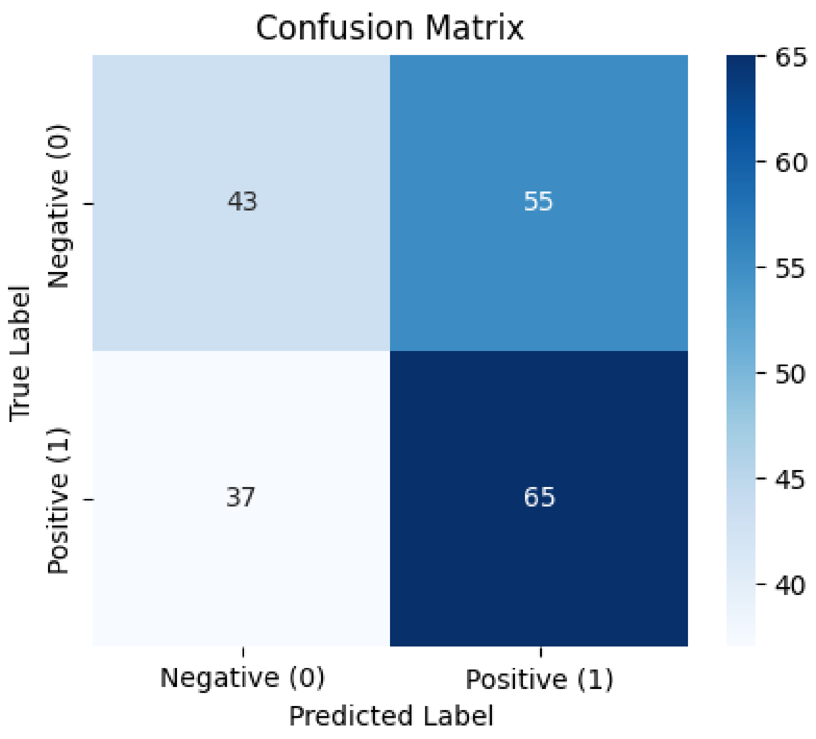
ROC Curve (Receiver Operating Characteristic Curve): The ROC Curve is a graphical representation used to evaluate the performance of a binary classification model across all classification thresholds. It plots two metrics: - True Positive Rate (TPR) or Recall (Sensitivity)-It is the proportion of actual 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.

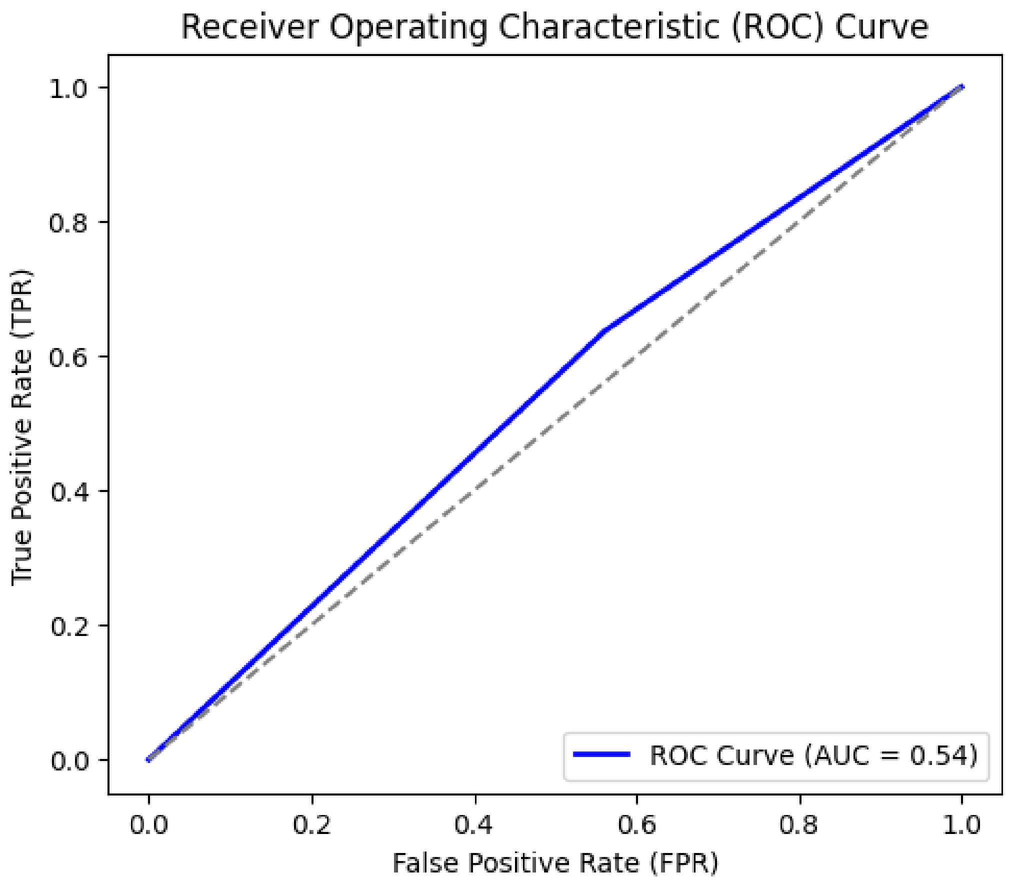
Classification Report: A Classification Report is a summary of various classification metrics, which are useful for evaluating the performance of a classifier on the given dataset.

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.

```
[ ]: ## Confusion Matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Compute confusion matrix
cm = confusion_matrix(y_test, test_preds)

# Plot confusion matrix
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Loss", "Win"],␣
 ↪yticklabels=["Loss", "Win"])
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```



[18]:
```
# Plot the ROC Curve & Calculate AUC Score
from sklearn.metrics import roc_curve, auc

# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test, test_preds)

# Compute AUC Score
roc_auc = auc(fpr, tpr)

# Plot ROC Curve
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color="blue", lw=2, label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], color="gray", linestyle="--")  # Random classifier line
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR)")
plt.title("Receiver Operating Characteristic (ROC) Curve")
plt.legend(loc="lower right")
plt.show()

# Print AUC score
print(f"AUC Score: {roc_auc:.4f}")
```



```
AUC Score: 0.5380
```

[20]:
```
# Generate a Classification Report
from sklearn.metrics import classification_report

# Generate classification report
report = classification_report(y_test, test_preds, target_names=['Loss', 'Win'])
print("Classification Report:\n", report)
```

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

        Loss       0.54      0.44      0.48        98
         Win       0.54      0.64      0.59       102

    accuracy                           0.54       200
   macro avg       0.54      0.54      0.53       200
weighted avg       0.54      0.54      0.54       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.

[22]:
```python
## Write your code here
# Save the model
torch.save(model.state_dict(), 'LogisticRegressionModel_v1.pth')

# Load the model
model2 = LogisticRegressionModel(input_units=X_train.shape[1], output_units=1)
model2.load_state_dict(torch.load('LogisticRegressionModel_v1.pth'))


# Ensure the loaded model is in evaluation mode
model2.eval()
with torch.no_grad():
   train_preds = model2(X_train)
   test_preds = model2(X_test)

train_preds = (train_preds >= 0.5).float()
test_preds = (test_preds >= 0.5).float()





# Evaluate the loaded model
train_accuracy = (train_preds.eq(y_train.view(-1, 1)).sum().item()) / y_train.
 ↪shape[0]
test_accuracy = (test_preds.eq(y_test.view(-1, 1)).sum().item()) / y_test.
 ↪shape[0]

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

```
Training Accuracy: 52.25%
Test Accuracy: 54.00%
```

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.

[24]:
```
from sklearn.metrics import accuracy_score

learning_rates = [0.01, 0.05, 0.1]

num_epochs = 100

best_lr = None
best_accuracy = 0.0
results = {}

for lr in learning_rates:
    print(f"\nTraining with Learning Rate: {lr}")

    # Reinitialize model and optimizer
    model = LogisticRegressionModel(input_units=X_train.shape[1],␣
 ↪output_units=1)
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    # Training loop
    for epoch in range(num_epochs):
        model.train()
        optimizer.zero_grad()

        outputs = model(X_train)
        loss = criterion(outputs, y_train.view(-1, 1))

        loss.backward()
        optimizer.step()


    # Evaluate test accuracy
    model.eval()
    with torch.no_grad():
        test_preds = model(X_test)
    test_preds = (test_preds >= 0.5).float()
    test_accuracy = accuracy_score(y_test, test_preds)

    # Store results
    results[lr] = test_accuracy
    print(f"Test Accuracy for Learning Rate {lr}: {test_accuracy:.4f}")

    # Track the best learning rate
    if test_accuracy > best_accuracy:
        best_accuracy = test_accuracy
        best_lr = lr

# Display final results
print("\n=== Final Learning Rate Comparison ===")
for lr, acc in results.items():
    print(f"Learning Rate: {lr} --> Test Accuracy: {acc:.4f}")

print(f"\nBest Learning Rate: {best_lr} with Test Accuracy: {best_accuracy:.
  ↪4f}")
```

```
Training with Learning Rate: 0.01
Test Accuracy for Learning Rate 0.01: 0.4200

Training with Learning Rate: 0.05
Test Accuracy for Learning Rate 0.05: 0.5550

Training with Learning Rate: 0.1
Test Accuracy for Learning Rate 0.1: 0.5700

=== Final Learning Rate Comparison ===
Learning Rate: 0.01 --> Test Accuracy: 0.4200
Learning Rate: 0.05 --> Test Accuracy: 0.5550
Learning Rate: 0.1 --> Test Accuracy: 0.5700

Best Learning Rate: 0.1 with Test Accuracy: 0.5700
```

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:

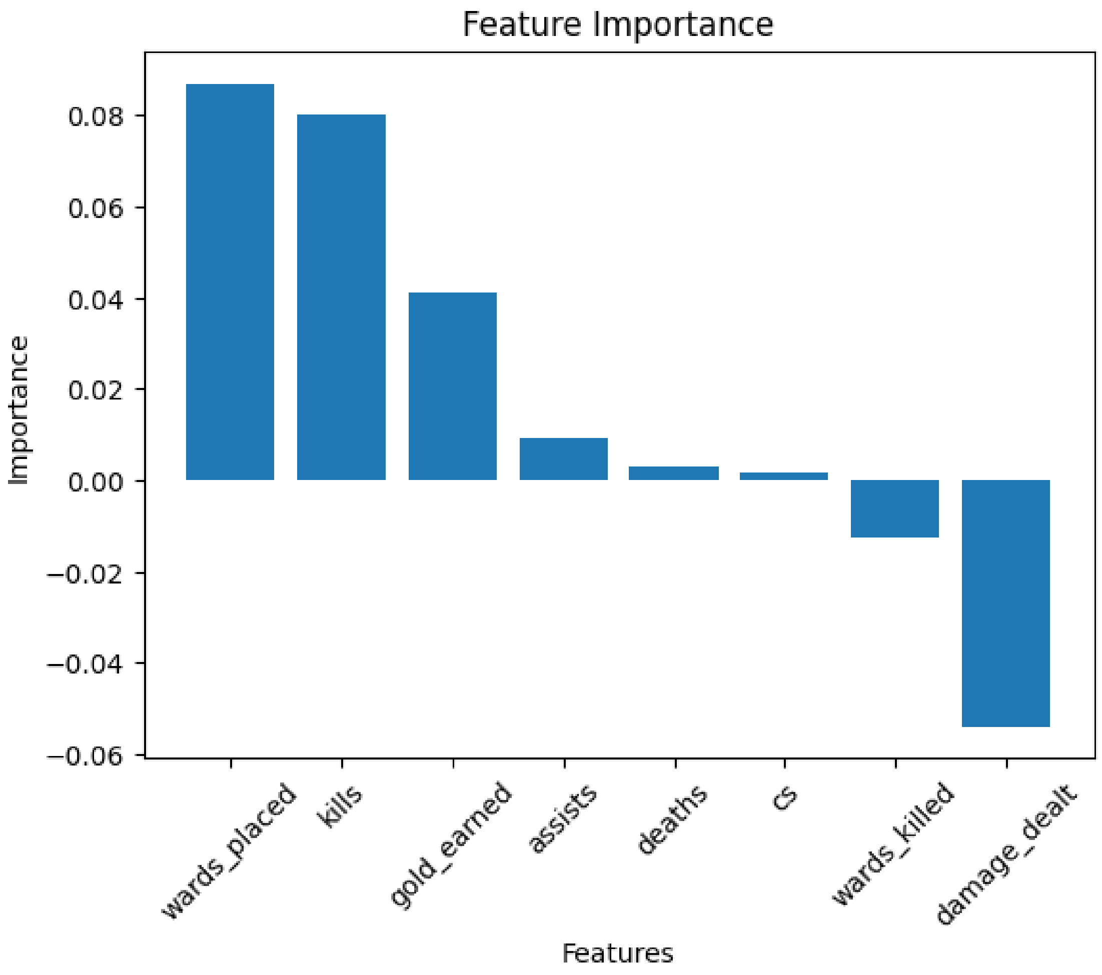
- Larger absolute weights indicate more influential features. Positive weights suggest a positive correlation with the outcome (likely to predict the positive class), while negative weights suggest the opposite.

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.

[26]:
```
## Write your code here

import pandas as pd
import matplotlib.pyplot as plt

# Extract the weights of the linear layer
weights = model.fc1.weight.data.numpy().flatten()
features = X.columns

display(weights)
display(features)

# Create a DataFrame for feature importance
feature_importance = pd.DataFrame({'Feature': features, 'Importance': weights})
feature_importance = feature_importance.sort_values(by='Importance',␣
 ↪ascending=False)
print(feature_importance)
# Plot feature importance plt.figure(figsize=(10, 6))
plt.bar(feature_importance['Feature'], feature_importance['Importance'])
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance')
plt.xticks(rotation=45)
plt.show()
```

```
array([ 0.08015746,  0.00313058,  0.00942263,  0.04131607,  0.00179946,
        0.08706514, -0.0127148 , -0.05395999], dtype=float32)
```

```
Index(['kills', 'deaths', 'assists', 'gold_earned', 'cs', 'wards_placed',
       'wards_killed', 'damage_dealt'],
      dtype='object')

        Feature  Importance
5  wards_placed    0.087065
0         kills    0.080157
3   gold_earned    0.041316
2       assists    0.009423
1        deaths    0.003131
4            cs    0.001799
6  wards_killed   -0.012715
7  damage_dealt   -0.053960
```



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.

```
[ ]:
```