In [1]:
```python
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
import os
```

In [2]:
```python
# Check if CUDA is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using GPU:", torch.cuda.get_device_name(0))
else:
    device = torch.device("cpu")
    print("CUDA is not available. Using CPU instead.")
```

```
Using GPU: NVIDIA GeForce RTX 4080
```

Data Preprocessing

In [3]:
```python
df = pd.read_pickle("./cdcdata.pkl")
```

One-hot encoding

In [4]:
```python
def one_hot_encode_features(df, columns_to_encode):
    for column in columns_to_encode:
        if column not in df.columns:
            raise ValueError(f"Column '{column}' not found in DataFrame.")

    df_encoded = pd.get_dummies(df, columns=columns_to_encode)

    return df_encoded

columns_to_encode = ['current_status', 'sex', 'age_group', 'race_ethnicity_combined', 'hosp_yn', 'icu_yn', 'death_yn', 'medcond_yn']
df_encoded = one_hot_encode_features(df, columns_to_encode)
```

Get rid of all columns related with date/time, and keep only one of any column with binary outcome

In [5]:
```python
cols_to_drop = ['cdc_case_earliest_dt', 'cdc_report_dt', 'pos_spec_dt', 'onset_dt', 'hosp_yn_No', 'icu_yn_No', 'death_yn_No', 'medcond_yn_No']
df_dropped = df_encoded.drop(cols_to_drop, axis=1)
df = df_dropped
```

In [6]:
```python
df # This is how dataframe looks
```

Out[6]:

	current_status_Laboratory-confirmed case	current_status_Probable Case	sex_Female	sex_Male	sex_Other	age_group_0 - 9 Years	age_group_10 - 19 Years	age_group_20 - 29 Years	age_group_30 - 39 Years	age_group_40 - 49 Years	...	race_ethnicity_combined_Asian, Non-Hispanic	race_ethnicity_combined_Black, Non-Hispanic	race_ethnicity_combined_Hispanic/Latino	race_ethnicity_combined_Multiple/Other, Non-Hispanic	race_ethnicity_combin Hawaiian/Other Pacifi Non
6	True	False	False	True	False	False	False	False	False	False	...	False	False	True	False	
11	True	False	False	True	False	False	False	False	False	False	...	False	False	True	False	
30	True	False	False	True	False	False	False	False	False	False	...	False	False	True	False	
36	True	False	False	True	False	False	False	False	False	False	...	False	False	True	False	
40	True	False	False	True	False	False	False	False	False	False	...	False	False	True	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
18336515	True	False	True	False	False	True	False	False	False	False	...	False	False	True	False	
18336519	True	False	True	False	False	True	False	False	False	False	...	False	False	True	False	
18336523	True	False	True	False	False	True	False	False	False	False	...	False	False	True	False	
18336525	True	False	True	False	False	True	False	False	False	False	...	False	False	True	False	
18336526	True	False	True	False	False	True	False	False	False	False	...	False	False	True	False	

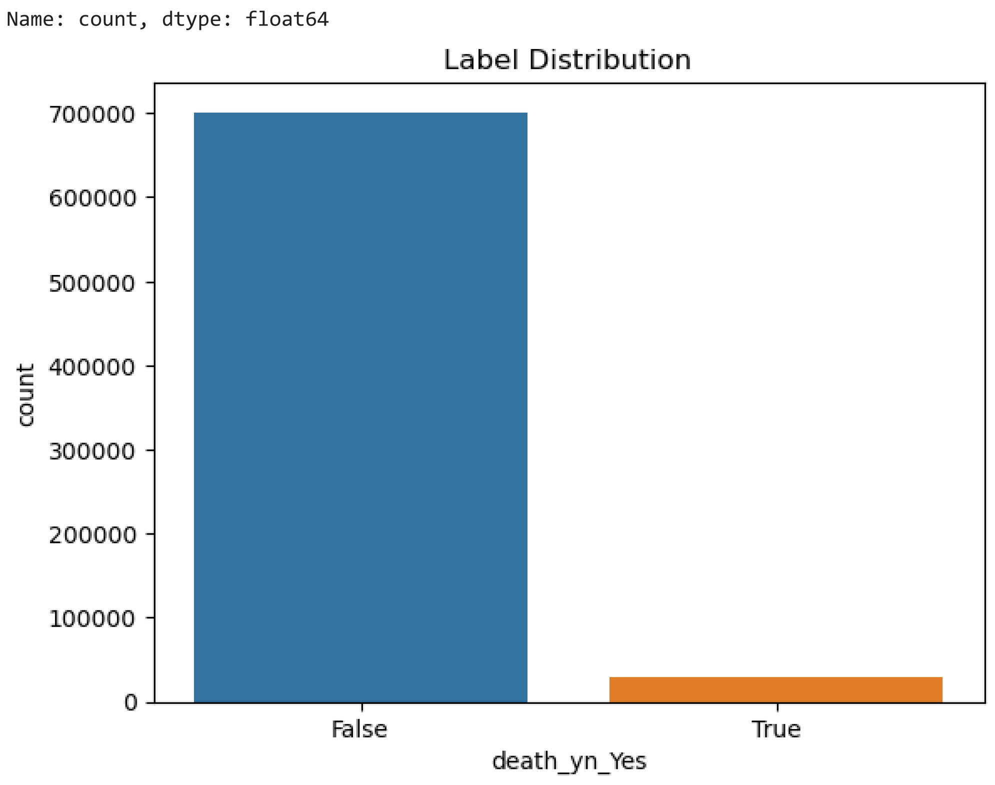
730187 rows × 25 columns

In [7]:
```python
label_counts = df['death_yn_Yes'].value_counts()
print(label_counts)

label_proportions = label_counts / len(df)
print(label_proportions)


sns.countplot(x=df['death_yn_Yes'])
plt.title('Label Distribution')
plt.show()
```

```
death_yn_Yes
False    701401
True      28786
Name: count, dtype: int64
death_yn_Yes
False    0.960577
True     0.039423
Name: count, dtype: float64
```



In [8]:
```python
features = df.drop('death_yn_Yes', axis=1)
labels = df['death_yn_Yes']
features_tensor = torch.tensor(features.values, dtype=torch.float32)
labels_tensor = torch.tensor(labels.values, dtype=torch.float32)
```

In [9]:
```python
X = features_tensor.numpy()
y = labels_tensor.numpy()

X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=1/9, random_state=42)  # making the split 8:1:1

X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)
```

In [10]:
```python
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

batch_size = 128

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

Model Architecture

In [13]:
```python
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_size):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(input_size, 1)

    def forward(self, x):
        logits = self.linear(x)
        probabilities = torch.sigmoid(logits)
        return probabilities
```

In [14]:
```python
input_size = 24
model = LogisticRegressionModel(input_size)
model = model.to(device)

loss_function = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

num_epochs = 100
```

In [15]:
```python
def train_model(model, train_loader, val_loader, loss_function, optimizer, num_epochs=10):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        for batch_features, batch_labels in train_loader:
            batch_features = batch_features.to(device)
            batch_labels = batch_labels.to(device)
            optimizer.zero_grad()
```

```
            outputs = model(batch_features)
            loss = loss_function(outputs.squeeze(), batch_labels.float())
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * batch_features.size(0)

        epoch_loss = running_loss / len(train_loader.dataset)

        # Evaluation on the validation set
        val_accuracy = evaluate_model(model, val_loader)

        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Validation Accuracy: {val_accuracy:.4f}')

    print("Training complete")


def evaluate_model(model, data_loader):
    model.eval()
    correct_predictions = 0
    total_predictions = 0

    with torch.no_grad():
        for batch_features, batch_labels in data_loader:
            batch_features = batch_features.to(device)
            batch_labels = batch_labels.to(device)
            outputs = model(batch_features)
            # predicted = torch.round(torch.sigmoid(outputs.squeeze()))
            predicted = torch.round(outputs.squeeze())
            correct_predictions += (predicted == batch_labels).sum().item()
            total_predictions += batch_labels.size(0)

    accuracy = correct_predictions / total_predictions
    return accuracy
```

In [16]:

```
train_model(model, train_loader, val_loader, loss_function, optimizer, num_epochs=10)

test_accuracy = evaluate_model(model, test_loader)
print(f'Test Accuracy: {test_accuracy:.4f}')
```

```
Epoch 1/10, Loss: 0.0870, Validation Accuracy: 0.9668
Epoch 2/10, Loss: 0.0818, Validation Accuracy: 0.9668
Epoch 3/10, Loss: 0.0817, Validation Accuracy: 0.9665
Epoch 4/10, Loss: 0.0818, Validation Accuracy: 0.9668
Epoch 5/10, Loss: 0.0817, Validation Accuracy: 0.9667
Epoch 6/10, Loss: 0.0817, Validation Accuracy: 0.9668
Epoch 7/10, Loss: 0.0818, Validation Accuracy: 0.9667
Epoch 8/10, Loss: 0.0818, Validation Accuracy: 0.9668
Epoch 9/10, Loss: 0.0817, Validation Accuracy: 0.9667
Epoch 10/10, Loss: 0.0817, Validation Accuracy: 0.9669
Training complete
Test Accuracy: 0.9678
```

Defining Better Metrics

Just an example illustrating how ROC-AUC works

In [18]:

```
import numpy as np
from sklearn.metrics import roc_auc_score
y_true = np.array([0]*950 + [1]*50)
y_pred = np.zeros_like(y_true)  # Predicts false for all

# Calculate the Accuracy
correct_predictions = np.sum(y_true == y_pred)
total_predictions = len(y_true)
accuracy = correct_predictions / total_predictions
print(f"Accuracy: {accuracy:.4f}")

# Calculate the AUC-ROC score
auc_roc_score = roc_auc_score(y_true, y_pred)
print(f"AUC-ROC Score: {auc_roc_score:.4f}")
```

```
Accuracy: 0.9500
AUC-ROC Score: 0.5000
```

In [19]:

```
def calculate_roc_score(model, test_loader):
    all_preds = []
    true_labels = []
    model.eval()
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs = inputs.to(device)

            outputs = model(inputs)

            all_preds.extend(outputs.squeeze().cpu().numpy())
            true_labels.extend(labels.cpu().numpy())
    auc_roc_score = roc_auc_score(true_labels, all_preds)
    print(f"AUC-ROC Score: {auc_roc_score:.4f}")
calculate_roc_score(model, test_loader)
```

```
AUC-ROC Score: 0.9652
```