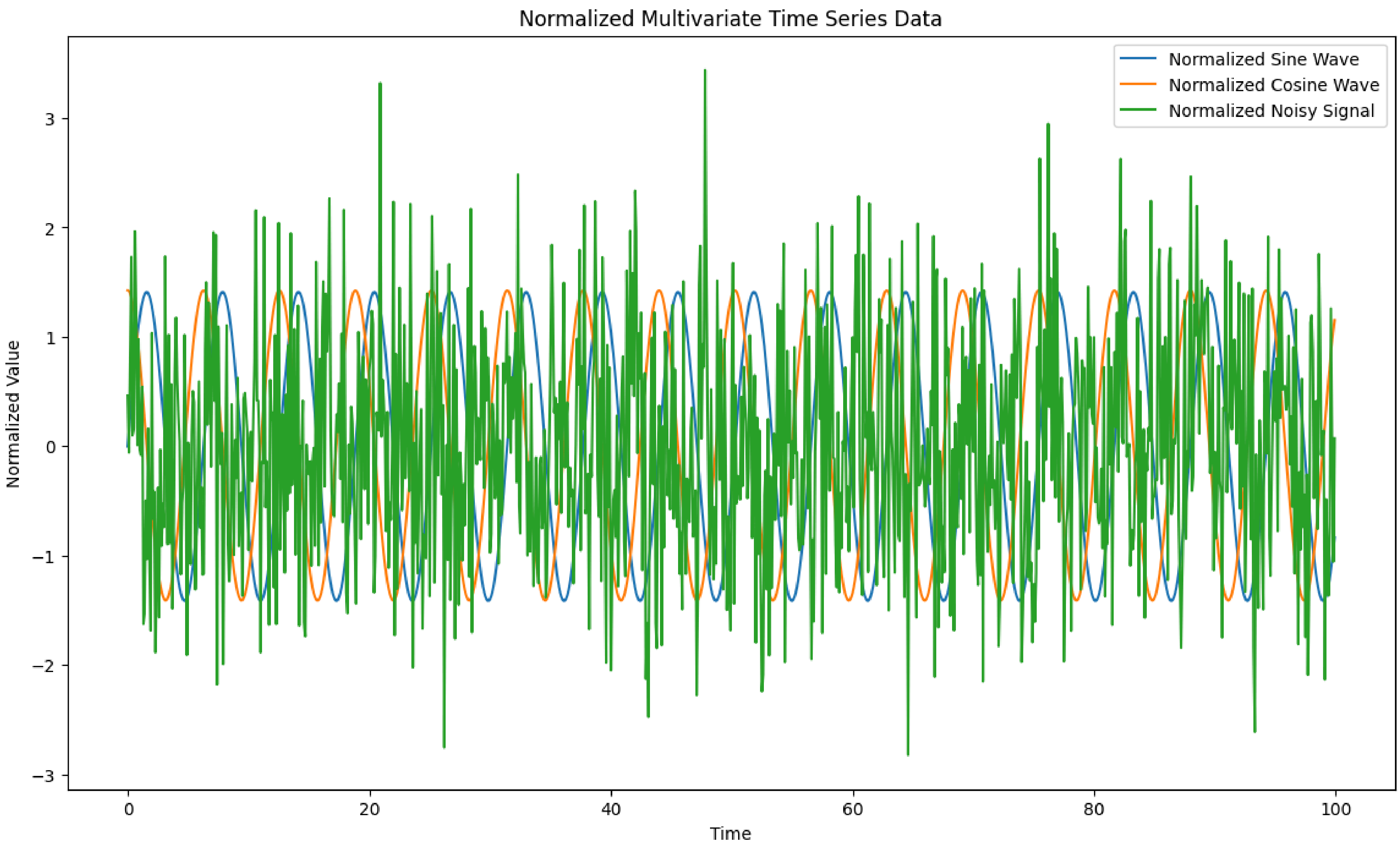
1.数据集生成以及预处理

In [15]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# 设置随机种子以保证结果可重复
np.random.seed(42)

# 生成时间序列数据
time_steps = np.arange(0, 100, 0.1)
sine_wave = np.sin(time_steps)
cosine_wave = np.cos(time_steps)
noisy_signal = np.random.normal(0, 1, len(time_steps)) + 0.5 * np.sin(2 * time_steps)

# 创建数据集
data = pd.DataFrame({
    'Time': time_steps,
    'Sine': sine_wave,
    'Cosine': cosine_wave,
    'Noisy': noisy_signal
})

# 数据标准化
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data[['Sine', 'Cosine', 'Noisy']])
scaled_data = pd.DataFrame(scaled_data, columns=['Sine', 'Cosine', 'Noisy'])
scaled_data['Time'] = data['Time']

# 可视化标准化后的数据
plt.figure(figsize=(14, 8))
plt.plot(scaled_data['Time'], scaled_data['Sine'], label='Normalized Sine Wave')
plt.plot(scaled_data['Time'], scaled_data['Cosine'], label='Normalized Cosine Wave')
plt.plot(scaled_data['Time'], scaled_data['Noisy'], label='Normalized Noisy Signal')
plt.title('Normalized Multivariate Time Series Data')
plt.xlabel('Time')
plt.ylabel('Normalized Value')
```

```
plt.legend()
plt.show()
```



2. 数据集拆分与窗口化

In [16]:
```
from sklearn.model_selection import train_test_split

# 定义滑动窗口函数
def create_sliding_window(data, window_size):
    X, y = [], []
    for i in range(len(data) - window_size):
        X.append(data[i:(i + window_size), :])
        y.append(data[(i + 1):(i + window_size + 1), :])  # 预测整个下一个窗口
    return np.array(X), np.array(y)

# 设置窗口大小
window_size = 10

# 创建滑动窗口数据
X, y = create_sliding_window(scaled_data[['Sine', 'Cosine', 'Noisy']].values, window_size)

# 拆分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3. Transformer 回归模型

In [20]:
```
import torch
import torch.nn as nn

class TransformerModel(nn.Module):
    def __init__(self, input_dim, output_dim, nhead, num_encoder_layers, dim_feedforward):
        super(TransformerModel, self).__init__()
        self.transformer_encoder = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=input_dim, nhead=nhead, dim_feedforward=dim_feedforward),
            num_layers=num_encoder_layers
        )
        self.fc = nn.Linear(input_dim, output_dim)

    def forward(self, src):
        # src的形状应该是 [window_size, batch_size, input_dim]
        output = self.transformer_encoder(src)
        # 对整个序列进行预测
        output = self.fc(output)
```

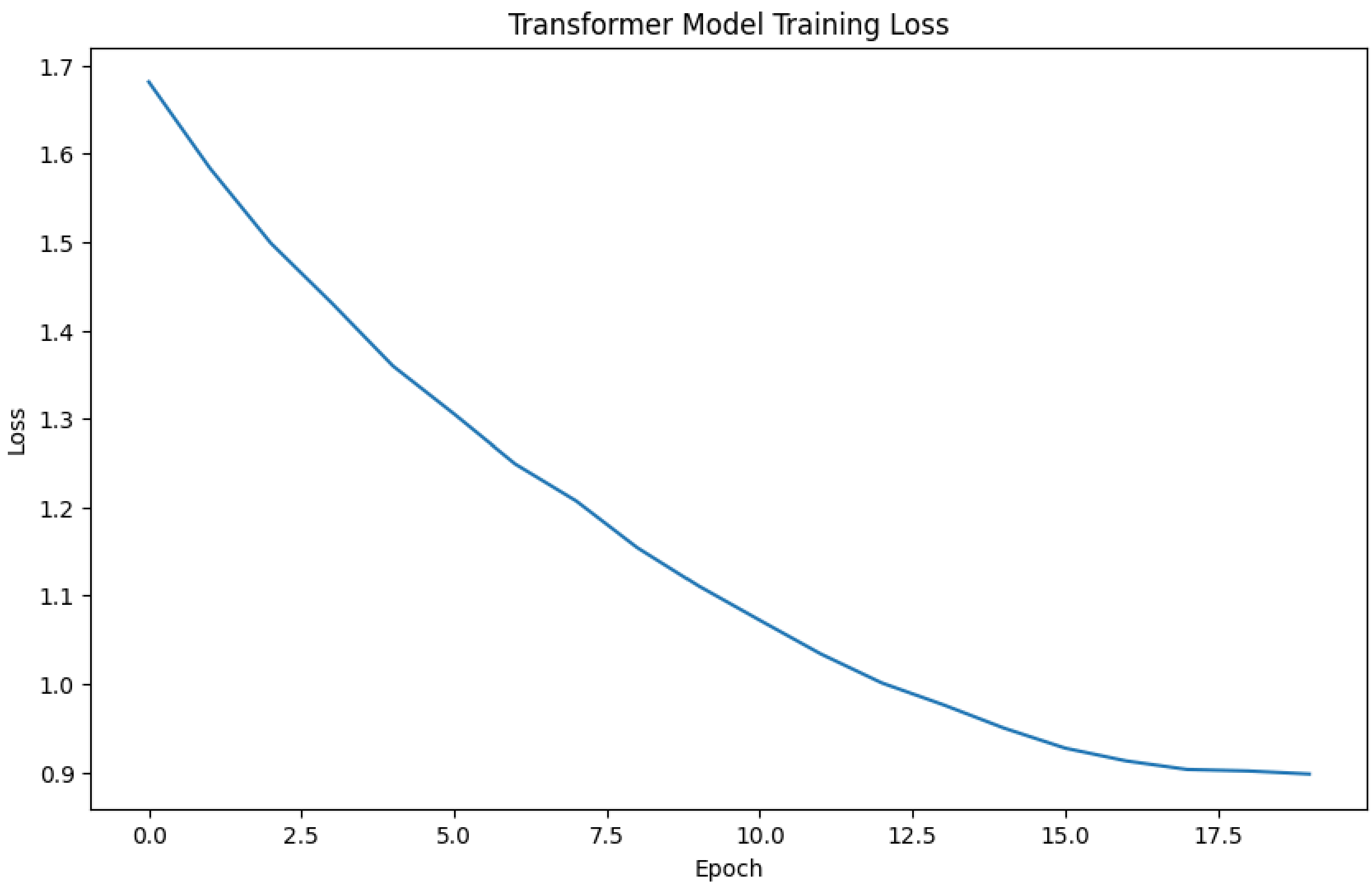
```
        return output

# 模型参数
input_dim = 3
output_dim = 3
nhead = 1
num_encoder_layers = 2
dim_feedforward = 512

# 初始化模型
model = TransformerModel(input_dim, output_dim, nhead, num_encoder_layers, dim_feedforward)
```

4.模型训练

In [21]:
```
import torch.optim as optim

# 转换数据为PyTorch张量
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# 定义损失函数和优化器
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 20
losses = []

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

    # 输入数据的维度调整为 [window_size, batch_size, input_dim] 以符合Transformer的输入要求
    outputs = model(X_train_tensor.permute(1, 0, 2))

    # 确保输出是三维的，然后调整维度顺序
    outputs = outputs.permute(1, 0, 2)

    # 计算损失，确保输出和目标的形状匹配 [batch_size, window_size, output_dim]
```

```
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()
    losses.append(loss.item())
    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}')

# 绘制训练损失曲线
plt.figure(figsize=(10, 6))
plt.plot(losses)
plt.title('Transformer Model Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```

```
Epoch 1/20, Loss: 1.6812819242477417
Epoch 2/20, Loss: 1.5836831331253052
Epoch 3/20, Loss: 1.4985456466674805
Epoch 4/20, Loss: 1.43062162399292
Epoch 5/20, Loss: 1.359494686126709
Epoch 6/20, Loss: 1.3052971363067627
Epoch 7/20, Loss: 1.2489795684814453
Epoch 8/20, Loss: 1.207135558128357
Epoch 9/20, Loss: 1.1542305946350098
Epoch 10/20, Loss: 1.1108592748641968
Epoch 11/20, Loss: 1.0720534324645996
Epoch 12/20, Loss: 1.0338760614395142
Epoch 13/20, Loss: 1.001030445098877
Epoch 14/20, Loss: 0.9764649271965027
Epoch 15/20, Loss: 0.9498901963233948
Epoch 16/20, Loss: 0.9272328019142151
Epoch 17/20, Loss: 0.9127388596534729
Epoch 18/20, Loss: 0.903052031993866
Epoch 19/20, Loss: 0.9013416767120361
Epoch 20/20, Loss: 0.8978870511054993
```



5.随机森林回归器

In [22]:
```
from sklearn.ensemble import RandomForestRegressor

# 将数据转换为2D格式以适应随机森林
X_train_rf = X_train.reshape(X_train.shape[0], -1)
X_test_rf = X_test.reshape(X_test.shape[0], -1)
```

```
y_train_rf = y_train.reshape(y_train.shape[0], -1)
y_test_rf = y_test.reshape(y_test.shape[0], -1)

# 初始化随机森林模型
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# 训练随机森林模型
rf_model.fit(X_train_rf, y_train_rf)

# 预测
rf_predictions = rf_model.predict(X_test_rf)
```

6.评估模型性能

In [23]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error

model.eval()
with torch.no_grad():
    # Transformer模型预测
    transformer_outputs = model(X_test_tensor.permute(1, 0, 2)).permute(1, 0, 2)
    transformer_outputs = transformer_outputs.numpy().reshape(y_test_rf.shape)
    transformer_mse = mean_squared_error(y_test_rf, transformer_outputs)
    transformer_mae = mean_absolute_error(y_test_rf, transformer_outputs)
    print(f'Transformer MSE: {transformer_mse}, MAE: {transformer_mae}')

    # 随机森林模型评估
    rf_mse = mean_squared_error(y_test_rf, rf_predictions)
    rf_mae = mean_absolute_error(y_test_rf, rf_predictions)
    print(f'Random Forest MSE: {rf_mse}, MAE: {rf_mae}')
```

```
Transformer MSE: 0.891311763629347, MAE: 0.7707247728399157
Random Forest MSE: 0.19194779737218376, MAE: 0.20440593091692713
```

7.加权平均融合

In [24]:
```
# 加权融合模型预测
alpha = 0.5  # 可以调整这个权重
ensemble_predictions = alpha * transformer_outputs + (1 - alpha) * rf_predictions
ensemble_mse = mean_squared_error(y_test_rf, ensemble_predictions)
```

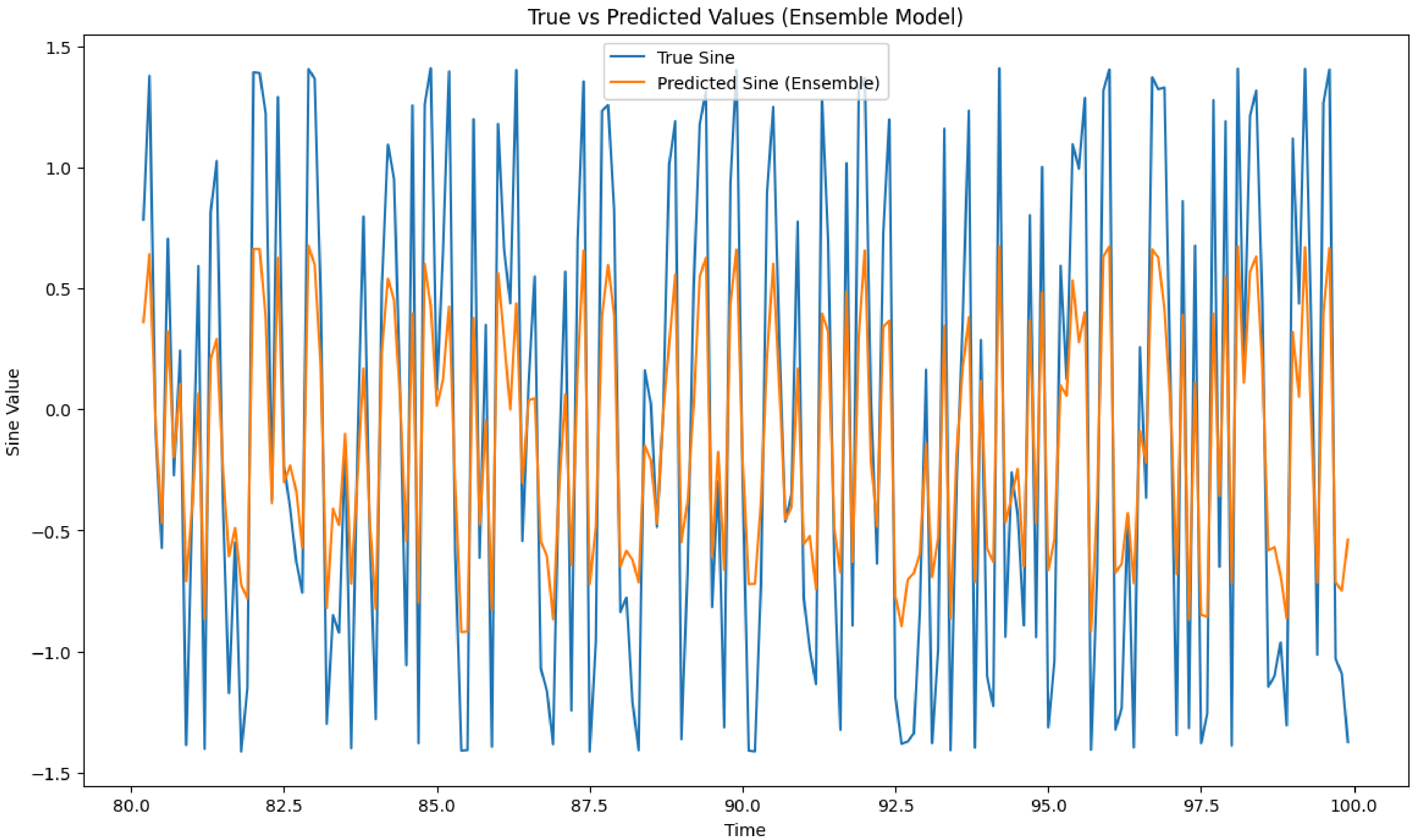
```
ensemble_mae = mean_absolute_error(y_test_rf, ensemble_predictions)
print(f'Ensemble Model MSE: {ensemble_mse}, MAE: {ensemble_mae}')
```

```
Ensemble Model MSE: 0.3909282047020254, MAE: 0.4768137310312925
```

8.真实与预测值对比图

In [25]:

```
# 可视化融合模型预测结果与真实值的对比
plt.figure(figsize=(14, 8))
plt.plot(data['Time'][len(data['Time']) - len(y_test):], y_test_rf[:, 0], label='True Sine')
plt.plot(data['Time'][len(data['Time']) - len(y_test):], ensemble_predictions[:, 0], label='Predicted Sine (Ensemble)')
plt.title('True vs Predicted Values (Ensemble Model)')
plt.xlabel('Time')
plt.ylabel('Sine Value')
plt.legend()
plt.show()
```



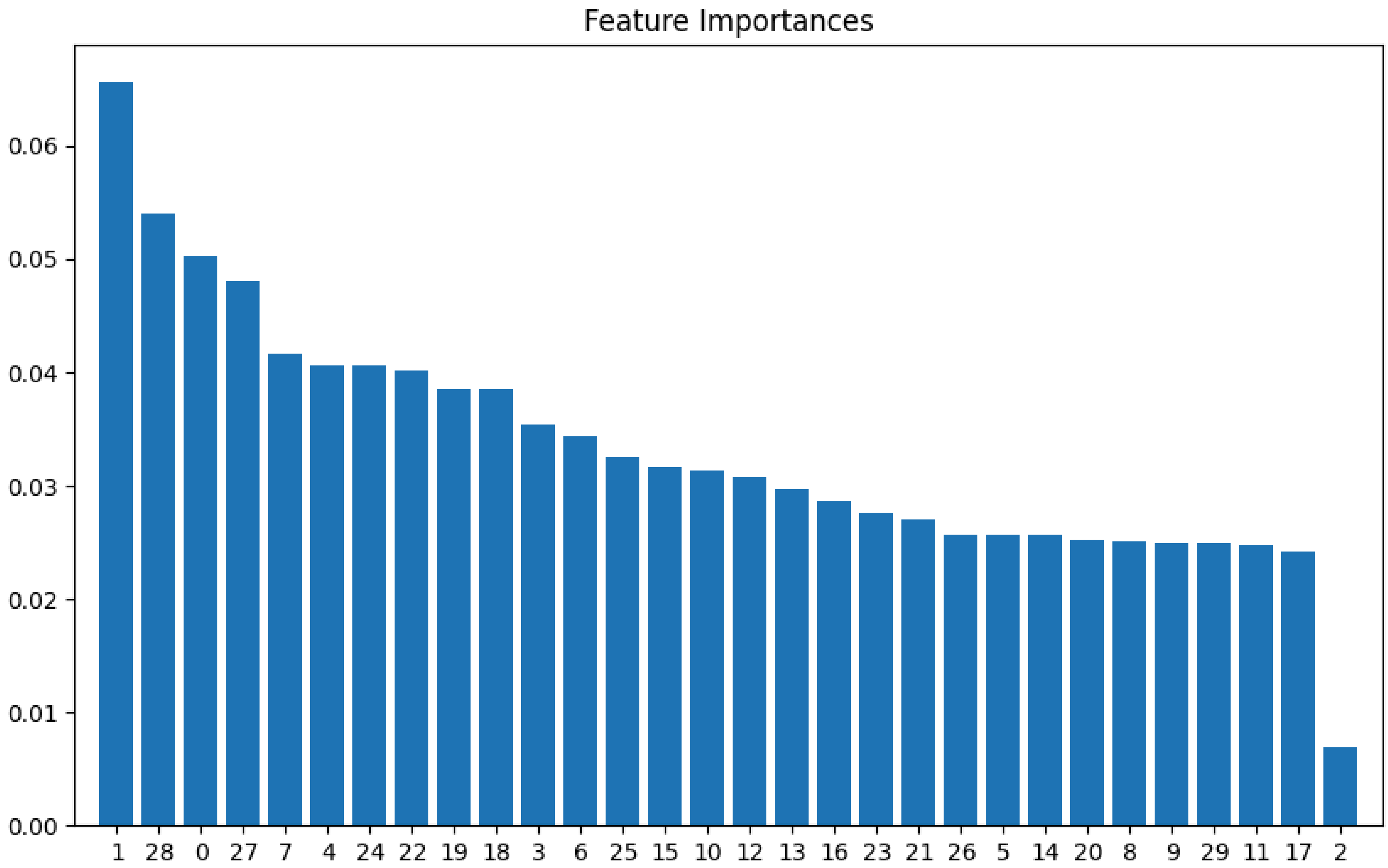
9.随机森林特征重要性

In [26]:

```
importances = rf_model.feature_importances_
indices = np.argsort(importances)[::-1]

# 打印特征重要性
print("Feature importances:")
```

```
for i in range(X_train_rf.shape[1]):
    print(f"{i+1}. feature {indices[i]} ({importances[indices[i]]})")

# 可视化特征重要性
plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.bar(range(X_train_rf.shape[1]), importances[indices], align="center")
plt.xticks(range(X_train_rf.shape[1]), indices)
plt.xlim([-1, X_train_rf.shape[1]])
plt.show()
```

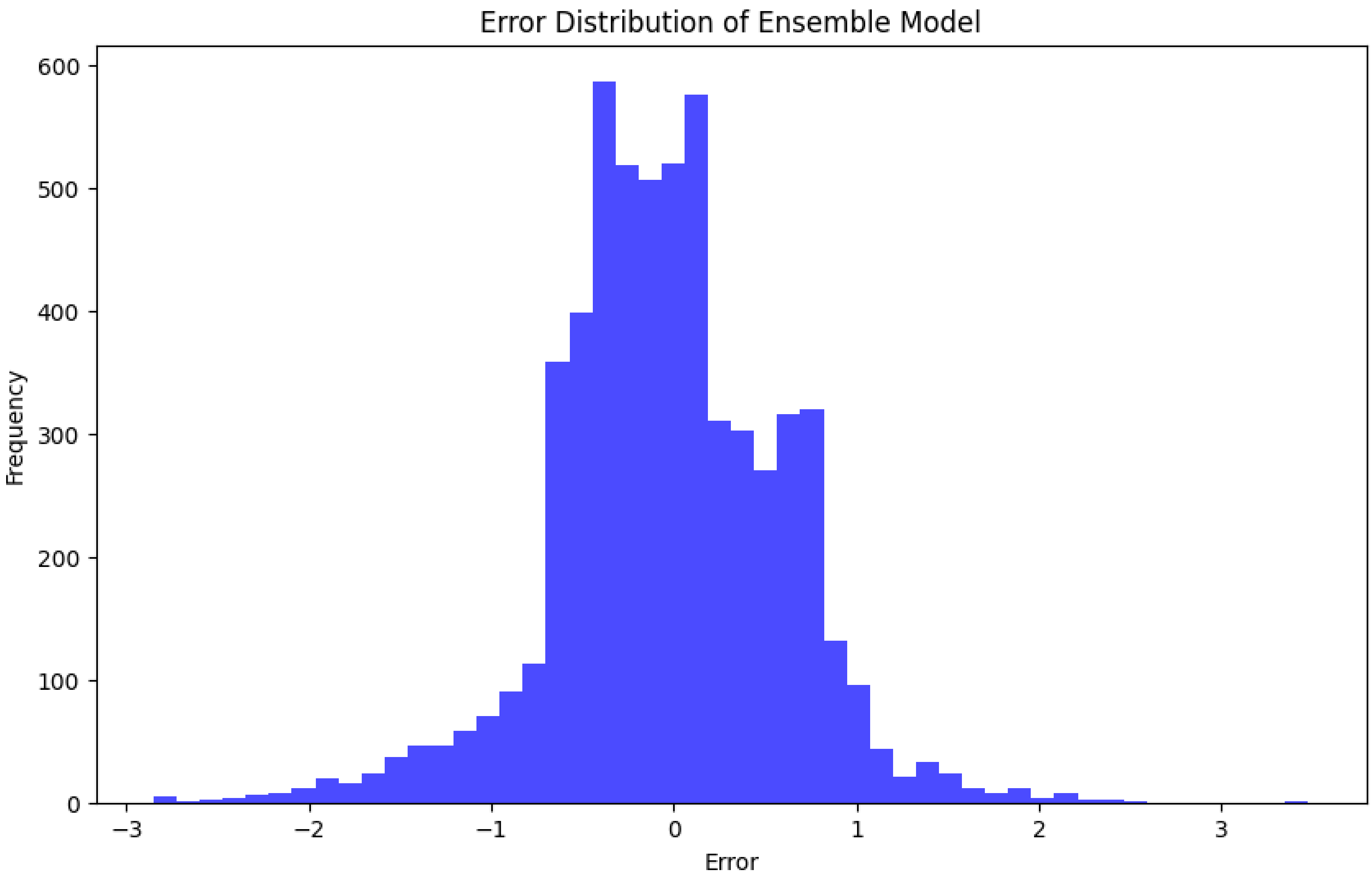
```
Feature importances:
1. feature 1 (0.06559879717305543)
2. feature 28 (0.05398579399665403)
3. feature 0 (0.05029070333147473)
4. feature 27 (0.04811832216022453)
5. feature 7 (0.041642686257272604)
6. feature 4 (0.04061001227858025)
7. feature 24 (0.04056937799644407)
8. feature 22 (0.04018158322105269)
9. feature 19 (0.038503057721303816)
10. feature 18 (0.03848534641095462)
11. feature 3 (0.03533301643206623)
12. feature 6 (0.03435143448126601)
13. feature 25 (0.032516390185344225)
14. feature 15 (0.03162136247098307)
15. feature 10 (0.03141999782517004)
16. feature 12 (0.030742591714885723)
17. feature 13 (0.029660940739293574)
18. feature 16 (0.028712798943951765)
19. feature 23 (0.027586249820303728)
20. feature 21 (0.027091894019428752)
21. feature 26 (0.02570156830099831)
22. feature 5 (0.025678888288781065)
23. feature 14 (0.02561870282844926)
24. feature 20 (0.025178658625207816)
25. feature 8 (0.025102312665502967)
26. feature 9 (0.024937803495908562)
27. feature 29 (0.024929082438157692)
28. feature 11 (0.024716973299423798)
29. feature 17 (0.024233428067606085)
30. feature 2 (0.006880224810254549)
```



10.融合模型误差分布

In [27]:
```
errors = y_test_rf - ensemble_predictions

plt.figure(figsize=(10, 6))
plt.hist(errors.flatten(), bins=50, alpha=0.7, color='blue')
plt.title('Error Distribution of Ensemble Model')
plt.xlabel('Error')
```

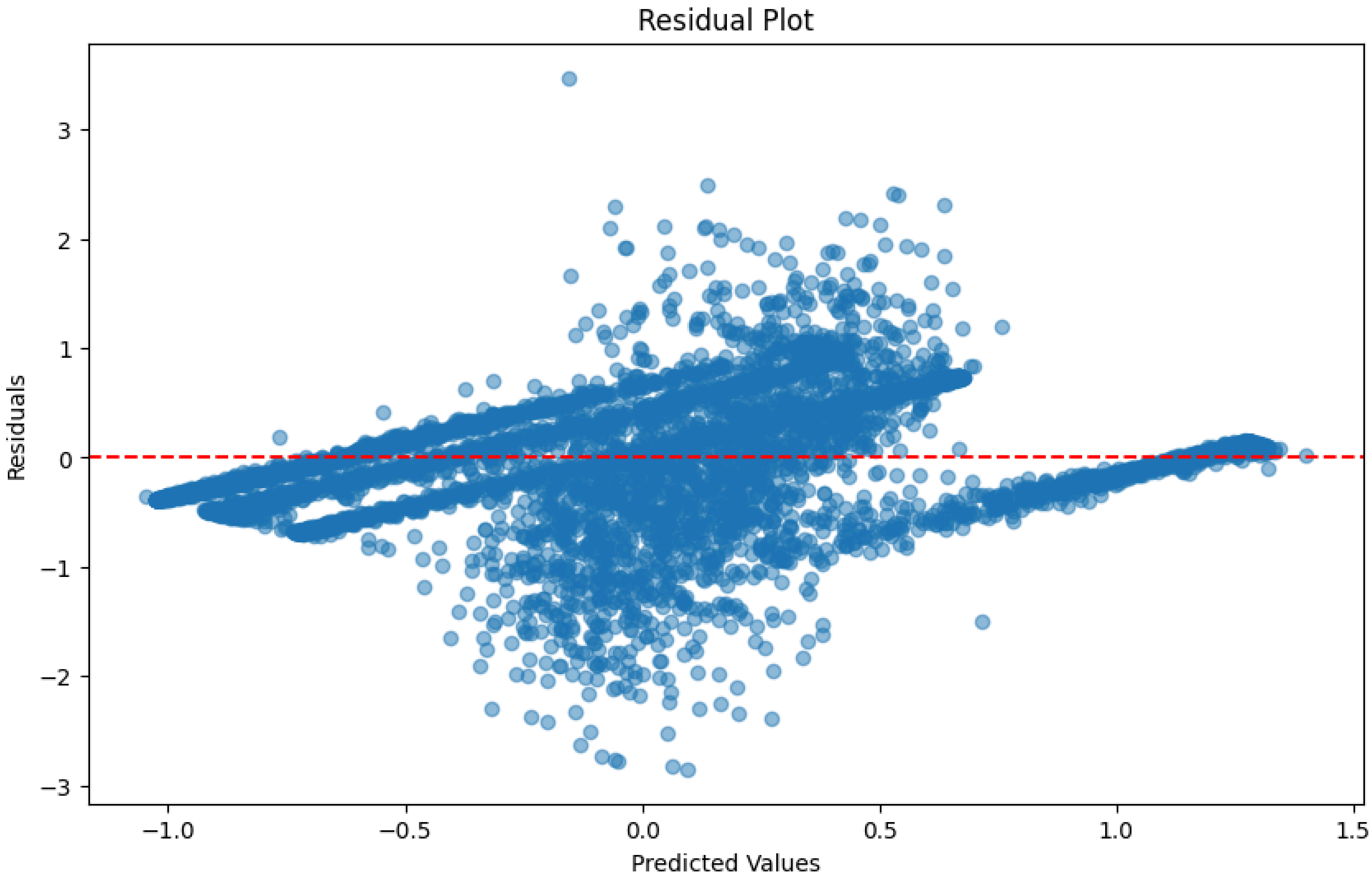
```
plt.ylabel('Frequency')
plt.show()
```



11.残差图

In [28]:
```
# 计算残差
residuals = y_test_rf - ensemble_predictions
```

```
# 绘制残差图
plt.figure(figsize=(10, 6))
plt.scatter(ensemble_predictions, residuals, alpha=0.5)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Residual Plot')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()
```



12.预测Vs真实值

In [29]:
```
# 绘制预测 vs 真实值图
plt.figure(figsize=(10, 6))
plt.scatter(y_test_rf, ensemble_predictions, alpha=0.5)
plt.plot([y_test_rf.min(), y_test_rf.max()], [y_test_rf.min(), y_test_rf.max()], 'k--', lw=2)
plt.title('Predicted vs Actual Plot')
```

```
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.show()
```