Q1e. Normalized Training and Test Error vs. Lambda

[3]:
```
rng=np.random.default_rng()
train_n=100
test_n=1000
d=100
mu=np.zeros(d)
```

[4]:
```
#Generate random dxd covariance matrix
A=np.random.rand(d,d)
while np.linalg.matrix_rank(A)!=d:
    A=np.random.rand(d,d)
Cov=A.T@A

sigma_noise=rng.uniform(0.3,0.7)
```

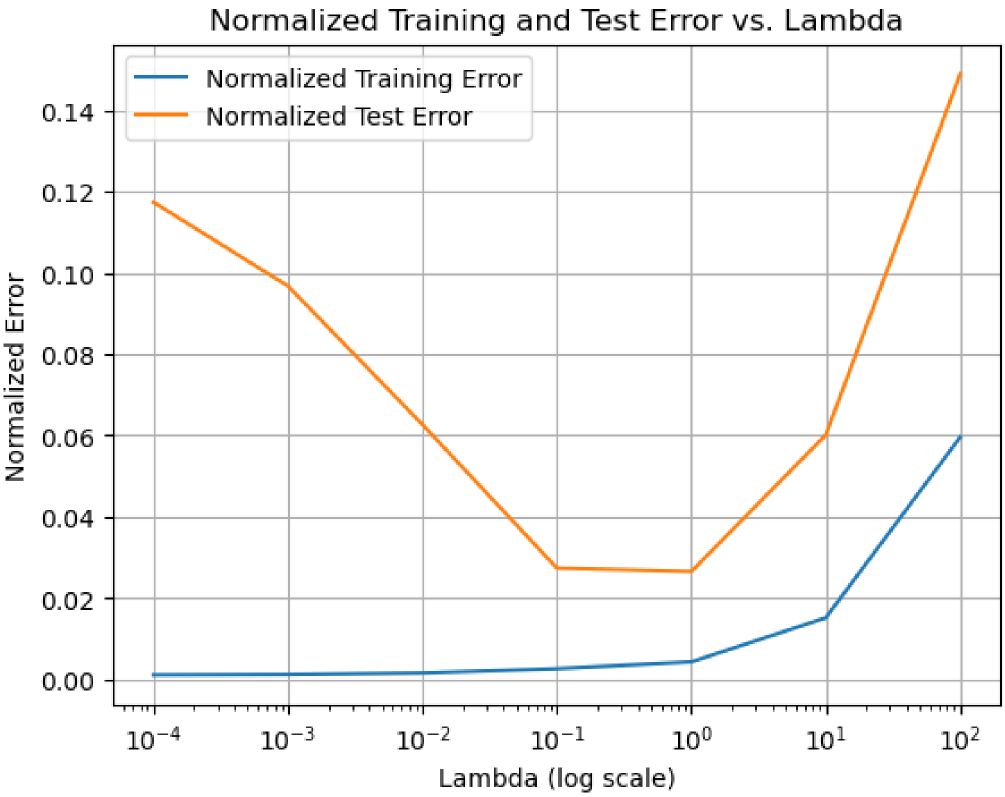
[5]:
```
#Generate training and test data
X_train=rng.multivariate_normal(mu,Cov,size=train_n)
a_true=rng.normal(0,1,size=(d,1))
y_train=X_train.dot(a_true)+np.random.normal(0,sigma_noise,size=(train_n,1))
X_test=rng.multivariate_normal(mu,Cov,size=test_n)
y_test=X_test.dot(a_true)+np.random.normal(0,sigma_noise,size=(test_n,1))
```

[13]:
```
lambda_values = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]
avg_train_errors = []
avg_test_errors = []
num_trials = 30
```

•[14]:
```
for lambda_val in lambda_values:
    train_error_sum = 0
    test_error_sum = 0

    for _ in range(num_trials):
        # Generate new training and test data with fixed parameters
        # Calculate the ridge regression solution
        XTX = np.dot(X_train.T, X_train)
        w = np.linalg.solve(XTX + lambda_val * np.identity(d), np.dot(X_train.T, y_train))

        # Normalized training error
        train_error = np.linalg.norm(np.dot(X_train, w) - y_train) / np.linalg.norm(y_train)

        # Normalized test error
        test_error = np.linalg.norm(np.dot(X_test, w) - y_test) / np.linalg.norm(y_test)

        train_error_sum += train_error
        test_error_sum += test_error

    # Average errors across trials for this lambda
    avg_train_error = train_error_sum / num_trials
    avg_test_error = test_error_sum / num_trials
    avg_train_errors.append(avg_train_error)
    avg_test_errors.append(avg_test_error)
```

[15]:
```python
plt.grid(True)
plt.semilogx(lambda_values, avg_train_errors, label='Normalized Training Error')
plt.semilogx(lambda_values, avg_test_errors, label='Normalized Test Error')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Normalized Error')
plt.title('Normalized Training and Test Error vs. Lambda')
plt.legend()
```

[15]:
```
<matplotlib.legend.Legend at 0x22dcda73e50>
```



Q2. Loading the Data

```
[77]: import numpy as np
      import torch, torchvision
      from sklearn.model_selection import train_test_split
      import matplotlib.pyplot as plt
```

```
[133]: train_set = torchvision.datasets.FashionMNIST("./data", download=True)
       test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False)
       X_train = train_set.data.numpy()
       labels_train = train_set.targets.numpy()
       X_test = test_set.data.numpy()
       labels_test = test_set.targets.numpy()
```

```
[134]: X_train = X_train.reshape((X_train.shape[0], X_train.shape[1]*X_train.shape[2]))
       X_test = X_test.reshape((X_test.shape[0], X_test.shape[1]*X_test.shape[2]))
```

```
[135]: X_train = X_train/255.0
       X_test = X_test/255.0
```

Q2b. Function to train the classifier

[155]:
```
def train(X, Y, lambda_):
    d = X.shape[1]
    k = Y.shape[1]  # Number of classes
    W = np.linalg.solve(X.T @ X + lambda_ * np.identity(d), X.T @ Y)
    return W
```

Function to predict labels

[137]:
```
def predict(W, X):
    return np.argmax(X @ W, axis=1)
```

[156]:
```
lambda_ = 1e-4
num_classes = 10
y = np.eye(num_classes)[labels_train]

# Train
W_hat = train(X_train, y, lambda_)
```

[157]:
```
predicted_labels_test = predict(W_hat, X_test)
```

[158]:
```
def evaluate_prediction(predictions, true_labels):
    num_errors = np.sum(predictions != true_labels)
    error_rate = num_errors / len(true_labels)
    return error_rate
```

[160]:
```
predicted_labels_train = predict(W_hat, X_train)

# Calculate training error
training_error = evaluate_prediction(predicted_labels_train, labels_train)
print("Training Error:", training_error)
```

```
Training Error: 0.17538333333333334
```

[159]:
```
predicted_labels_train = predict(W_hat, X_train)

# Calculate test error
training_error = evaluate_prediction(predicted_labels_test, labels_test)
print("Testing Error:", training_error)
```

```
Testing Error: 0.1913
```

Q2c. Partitioning and Plotting $\hat{W}^p$ vs p

[148]:
```
# Parameters
num_p_values = 10  # Number of different p values to try
max_p = 6000  # Maximum value of p to try
p_values = np.linspace(100, max_p, num_p_values, dtype=int)

training_errors = []
validation_errors = []
```

•[162]:
```
# Spliting Training data
X_train_split, X_validation, labels_train_split, labels_val = train_test_split(X_train, labels_train, test_size=0.2, random_state=42)
```

[161]:
```
for p in p_values:
    # Generate random matrix G with Gaussian entries
    G = np.random.normal(0, np.sqrt(0.005), size=(p, X_train_split.shape[1]))

    # Generate random vector b with uniform entries
    b = np.random.uniform(0, 2*np.pi, size=p)

    # Transform the training and validation data
    X_train_transformed = np.cos(G @ X_train_split.T + b[:, np.newaxis])
    X_val_transformed = np.cos(G @ X_validation.T + b[:, np.newaxis])

    # One-hot encode the training labels
    Y_train_encode = np.eye(num_classes)[labels_train_split]

    # Train the classifier
    W_hat = train(X_train_transformed.T, Y_train_encode, lambda_)

    # Predict labels for training and validation data
    predicted_labels_train = predict(W_hat, X_train_transformed.T)
    predicted_labels_val = predict(W_hat, X_val_transformed.T)

    # Calculate training and validation errors
    training_error = evaluate_prediction(predicted_labels_train, labels_train_split)
    validation_error = evaluate_prediction(predicted_labels_val, labels_val)

    training_errors.append(training_error)
    validation_errors.append(validation_error)
```

Plotting $\hat{W}^p$ vs p

```
[151]: plt.grid(True)
       plt.plot(p_values, training_errors, label='Training Error')
       plt.plot(p_values, validation_errors, label='Validation Error')
       plt.xlabel('p (Number of Transformed Features)')
       plt.ylabel('Error Rate')
       plt.title('Classification Error vs. p')
       plt.legend()
```

```
[151]: <matplotlib.legend.Legend at 0x199977f55d0>
```

Q2d. Computing the confidence interval

Transform the test data using the optimal p

[125]:
```
p_optimal = p_values[np.argmin(validation_errors)]
print("p_optimal: ", p_optimal)
```

```
p_optimal:  6000
```

•[126]:
```
G_optimal = np.random.normal(0, np.sqrt(0.005), size=(p_optimal, X_train.shape[1]))
b_optimal = np.random.uniform(0, 2*np.pi, size=p_optimal)
X_test_transformed = np.cos(G_optimal @ X_test.T + b_optimal[:, np.newaxis])
```

•[127]:
```
# Train the classifier using p̂ and Predict for test data
W_hat_optimal = train(X_train_transformed.T, Y_train_encode, lambda_)
predicted_labels_test_optimal = predict(W_hat_optimal, X_test_transformed.T)
print("predicted_labels_test_optimal: ", predicted_labels_test_optimal)

# Calculate the classification test error ϵ̂_test(f̂)
test_error_optimal = evaluate_prediction(predicted_labels_test_optimal, labels_test)
```

```
predicted_labels_test_optimal:  [9 2 9 ... 9 6 2]
```

•[128]:
```
confidence_level = 0.95
delta = 1 - confidence_level
a = 0  # Min of true classification error (0%)
b = 1  # Max of true classification error (100%)
m = len(labels_test)  # Number of test examples
```

Compute the confidence interval using Hoeffding's inequality

[153]:
```
confidence_interval = np.sqrt(((b - a)**2 * np.log(2/delta)) / (2 * m))
print("confidence_interval: ", confidence_interval)
```

```
confidence_interval:  [0.01355542 0.00932347 0.0706512  ... 0.03441607 0.01300412 0.03359333]
```

[154]:
```
lower_bound = test_error_optimal - confidence_interval
upper_bound = test_error_optimal + confidence_interval

print("lower_bound: ", lower_bound)
print("upper_bound: ", upper_bound)
print("Classification Test Error: ", test_error_optimal)
print("confidence_level : ", confidence_level)
```

```
lower_bound:  [0.88194458 0.88617653 0.8248488  ... 0.86108393 0.88249588 0.86190667]
upper_bound:  [0.90905542 0.90482347 0.9661512  ... 0.92991607 0.90850412 0.92909333]
Classification Test Error:  0.8955
confidence_level :  0.95
```