1. Data Normalization/Standardization

Sometimes, our features have very different ranges of values. This is not ideal and can lead to numerical issues (e.g., overflow) and optimization difficulties.

There are two ways to take care of this issue. One is called data normalization, and the other is called data standardization. Sometimes these terms are used interchangeably, but it is important to understand the difference.

Below, $x_i^{(j)}$ represents the value of the i^{th} feature of the j^{th} data point.

1.1. Data Standardization

Data standardization is the task of transforming each feature in our dataset to have mean 0 and variance 1. The typical way to do this is using the Z-Score, which is defined as below:

$$\tilde{x_i}^{(j)} = \frac{x_i^{(j)} - \mu_i}{\sigma_i}$$

Where μ_i is the mean of each feature and σ_i is the standard deviation of each feature, which are empirically calculated from the data.

Question: what should you do when $\sigma_i = 0$ for some i?

1.2. Data Normalization

Data normalization refers to the task of rescaling each feature in our dataset to have range [0, 1].

One such method to achieve this is min-max scaling:

$$\tilde{x_i}^{(j)} = \frac{x_i^{(j)} - x_i^{min}}{x_i^{max} - x_i^{min}}$$

Where x_i^{min}, x_i^{max} are the minimum and maximum values of feature i in our dataset, respectively.

When training and evaluating your model, you should calculate the parameters for your normalization or standardization function on the training set **ONLY**!

In other words, if we were using Z-Score, we'd calculate our μ_i, σ_i on the training set, and use those same values when standardizing our validation/test data. The same applies to normalization methods, such as min-max scaling, where we want to determine x_i^{min}, x_i^{max} from training data. This will likely mean that we may get some values outside [0, 1] after rescaling new data, but, given that our training dataset is large enough, this shouldn't be much of an issue (so unseen data likely won't be wildly outside of [0, 1]).

2. Regularization

Just a reminder, don't ever regularize your bias term. This term doesn't add any complexity to the model (since it just shifts), so we'd like it to take on any value that best fits our training data.

3. Biased Test Error

Is the test error unbiased for these programs? If not, how can we fix the code so it is?

3.1. Program 1

```
# Given dataset of 1000-by-50 feature
# matrix X, and 1000-by-1 labels vector

mu = np.mean(X, axis=0)
X = X - mu

idx = np.random.permutation(1000)
TRAIN = idx[0:900]
TEST = idx[900::]

ytrain = y[TRAIN]
Xtrain = X[TRAIN, :]

# solve for argmin_w ||Xtrain*w - ytrain||_2
w = np.linalg.solve(np.dot(Xtrain.T, Xtrain), np.dot(Xtrain.T, ytrain))

b = np.mean(ytrain)

ytest = y[TEST]
Xtest = X[TEST, :]

train_error = np.dot(np.dot(Xtrain, w)+b - ytrain,
              np.dot(Xtrain, w)+b - ytrain ) / len(TRAIN)
test_error = np.dot(np.dot(Xtest, w)+b - ytest,
              np.dot(Xtest, w)+b - ytest ) / len(TEST)

print('Train error = ', train_error)
print('Test error = ', test_error)
```

3.2. Program 2

```
# Given dataset of 1000-by-50 feature
# matrix X, and 1000-by-1 labels vector

idx = np.random.permutation(1000)
TRAIN = idx[0:900]
TEST = idx[900::]

ytrain = y[TRAIN]
Xtrain = X[TRAIN, :]
Xtrain_avg = np.mean(Xtrain, axis=0)
Xtrain = Xtrain - Xtrain_avg

# solve for argmin_w ||Xtrain*w - ytrain||_2
w = np.linalg.solve(np.dot(Xtrain.T, Xtrain), np.dot(Xtrain.T, ytrain))

b = np.mean(ytrain)

ytest = y[TEST]
Xtest = X[TEST, :]
Xtest_avg = np.mean(Xtest, axis=0)
Xtest = Xtest - Xtest_avg

train_error = np.dot(np.dot(Xtrain, w)+b - ytrain,
               np.dot(Xtrain, w)+b - ytrain ) / len(TRAIN)
test_error = np.dot(np.dot(Xtest, w)+b - ytest,
               np.dot(Xtest, w)+b - ytest ) / len(TEST)

print('Train error = ', train_error)
print('Test error = ', test_error)
```

3.3. Program 3

```
# Given dataset of 1000-by-50 feature
# matrix X, and 1000-by-1 labels vector

def fit(Xin, Yin):
    mu = np.mean(Xin, axis=0)
    Xin = Xin - mu
    w = np.linalg.solve(np.dot(Xin.T, Xin), np.dot(Xin.T, Yin))
    b = np.mean(Yin) - np.dot(w, mu)
    return w, b

def predict(w, b, Xin):
    return np.dot(Xin, w) + b

idx = np.random.permutation(1000)
TRAIN = idx[0:800]
VAL = idx[800:900]
TEST = idx[900::]

ytrain = y[TRAIN]
Xtrain = X[TRAIN, :]
yval = y[VAL]
Xval = X[VAL, :]

err = np.zeros(50)

# use cross validation to pick the best features to use
for d in range(1,51):
    w, b = fit(Xtrain[:, 0:d], ytrain)
    yval_hat = predict(w, b, Xval[:,0:d])
    err[d-1] = np.mean((yval_hat - yval)**2)

d_best = np.argmin(err) + 1

Xtot = np.concatenate((Xtrain, Xval), axis=0)
ytot = np.concatenate((ytrain, yval), axis=0)

w, b = fit(Xtot[:, 0:d_best], ytot)

ytest = y[TEST]
Xtest = X[TEST, :]

ytot_hat = predict(w, b, Xtot[:, 0:d_best])
train_error = np.mean((ytot_hat - ytot) **2)
ytest_hat = predict(w, b, Xtest[:, 0:d_best])
test_error = np.mean((ytest_hat - ytot) **2)

print('Train error = ', train_error)
print('Test error = ', test_error)
```

4. Extra: K-fold Cross-Validation (Demonstrative code)

```
# Given dataset of 1000-by-50 feature
# matrix X, and 1000-by-1 labels vector
import np


def fit(Xin, Yin, lbda):
      mu = np.mean(Xin, axis=0)
      Xin = Xin - mu
      w = np.linalg.solve(np.dot(Xin.T, Xin) + lbda, np.dot(Xin.T, Yin))
      b = np.mean(Yin) - np.dot(w, mu)
      return w, b


def predict(w, b, Xin):
      return np.dot(Xin, w) + b


# Note: X, y are all the train data and label for the entire experiments
N_TRAIN = 1000
idx = np.random.permutation(N_TRAIN)
K_FOLD = 5

NON_TEST = idx[0: 9 * N_TRAIN // 10]
N_PER_FOLD = len(NON_TEST) // K_FOLD
TEST = idx[9 * N_TRAIN // 10::]

# candidates regularization coefficient to choose from
lbdas = [0.1, 0.2, 0.3]
err = np.zeros(len(lbdas))

for lbda_idx, lbda in enumerate(lbdas):
      for i in range(K_FOLD):
            # CRUCIAL: we use slicing to calculate the index train set and val set should use!
            # Using the ith fold as the validation set
            VAL = NON_TEST[i * N_PER_FOLD:(i+1) * N_PER_FOLD]
            # Using the rest as the train set
            TRAIN = NON_TEST[:i * N_PER_FOLD] + NON_TEST[(i + 1) * N_PER_FOLD:]

            ytrain = y[TRAIN]
            Xtrain = X[TRAIN]
            yval = y[VAL]
            Xval = X[VAL]

            w, b = fit(Xtrain, ytrain, lbda)
            yval_hat = predict(w, b, Xval)
            # accumulate error from this fold of validation set
            err[lbda_idx] += np.mean((yval_hat - yval)**2)

      # calculate the error for the k-fold validation
      err[lbda_idx] /= K_FOLD

lbda_best = lbdas[np.argmin(err)]

Xtot = np.concatenate((Xtrain, Xval), axis=0)
ytot = np.concatenate((ytrain, yval), axis=0)

w, b = fit(Xtot, ytot, lbda_best)

```

```
    ytest = y[TEST]
    Xtest = X[TEST]

    ytot_hat = predict(w, b, Xtot)
    train_error = np.mean((ytot_hat - ytot) ** 2)
    ytest_hat = predict(w, b, Xtest)
    test_error = np.mean((ytest_hat - ytest) ** 2)

    print('Best choice of lambda = ', lbda_best)
    print('Train error = ', train_error)
    print('Test error = ', test_error)
```