

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 ?

Solution:

Having $\sigma_i = 0$ for some feature means that the value for that feature is constant in our dataset. If we leave it as 0, we will encounter a divide by 0 error. Since the feature is constant, once we subtract the mean, the new value for the feature will be 0, so we can divide by anything except 0 to avoid this error.

Having $\sigma_i = 0$ is rare, and may be a sign something is wrong with your data or code. One specific case to watch out for is appending your bias column of ones before standardizing.

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

```
1 # Given dataset of 1000-by-50 feature
2 # matrix X, and 1000-by-1 labels vector
3
4 mu = np.mean(X, axis=0)
5 X = X - mu
6
7 idx = np.random.permutation(1000)
8 TRAIN = idx[0:900]
9 TEST = idx[900:]
10
11 ytrain = y[TRAIN]
12 Xtrain = X[TRAIN, :]
13
14 # solve for argmin_w ||Xtrain*w - ytrain||_2
15 w = np.linalg.solve(np.dot(Xtrain.T, Xtrain), np.dot(Xtrain.T, ytrain))
16
17 b = np.mean(ytrain)
18
19 ytest = y[TEST]
20 Xtest = X[TEST, :]
21
22 train_error = np.dot(np.dot(Xtrain, w)+b - ytrain,
23                      np.dot(Xtrain, w)+b - ytrain) / len(TRAIN)
24 test_error = np.dot(np.dot(Xtest, w)+b - ytest,
25                    np.dot(Xtest, w)+b - ytest) / len(TEST)
26
27 print('Train error = ', train_error)
28 print('Test error = ', test_error)
```

Solution:

The error is at the beginning of the program on lines 4 and 5. Notice how μ is a function of both the train and test data. By de-meaning the entire dataset before splitting, we are intertwining the train and test data. The correct procedure is:

- Split into train and test
- Compute the mean of the train data, μ_{train}
- De-mean both the train and test data with μ_{train}

3.2. Program 2

```
1 # Given dataset of 1000-by-50 feature
2 # matrix X, and 1000-by-1 labels vector
3
4 idx = np.random.permutation(1000)
5 TRAIN = idx[0:900]
6 TEST = idx[900:]
7
8 ytrain = y[TRAIN]
9 Xtrain = X[TRAIN, :]
10 Xtrain_avg = np.mean(Xtrain, axis=0)
11 Xtrain = Xtrain - Xtrain_avg
12
13 # solve for argmin_w ||Xtrain*w - ytrain||_2
14 w = np.linalg.solve(np.dot(Xtrain.T, Xtrain), np.dot(Xtrain.T, ytrain))
15
16 b = np.mean(ytrain)
17
18 ytest = y[TEST]
19 Xtest = X[TEST, :]
20 Xtest_avg = np.mean(Xtest, axis=0)
21 Xtest = Xtest - Xtest_avg
22
23 train_error = np.dot(np.dot(Xtrain, w)+b - ytrain,
24                      np.dot(Xtrain, w)+b - ytrain ) / len(TRAIN)
25 test_error = np.dot(np.dot(Xtest, w)+b - ytest,
26                     np.dot(Xtest, w)+b - ytest ) / len(TEST)
27
28 print('Train error = ', train_error)
29 print('Test error = ', test_error)
```

Solution:

Notice that on line 21, we are demeaning the test set with the mean of the test set, instead of the mean of the training set. The w , b is trained on the demeaning of training set. Since we demean training data using the training set, we have to treat the test set identically to the way we treat the training set. Therefore, when making predictions, the demeaning is still part of the preprocessing data step, just like how we preprocess the training set. Therefore, we need to make sure the demean the data using the training set.

An alternate intuition for not demeaning using test set is to suppose we have 3 data points in the test set x_1 , x_2 , and x_3 . If x_1 and x_2 is a test set that we are making predictions on, the outcome of these prediction may be different from if we choose x_2 and x_3 to be the test set. This means that for the same datapoint, your outcome will be different depending on all the other points you have in the test set, which is incorrect.

3.3. Program 3

```
1 # Given dataset of 1000-by-50 feature
2 # matrix X, and 1000-by-1 labels vector
3
4 def fit(Xin, Yin):
5     mu = np.mean(Xin, axis=0)
6     Xin = Xin - mu
7     w = np.linalg.solve(np.dot(Xin.T, Xin), np.dot(Xin.T, Yin))
8     b = np.mean(Yin) - np.dot(w, mu)
9     return w, b
10
11 def predict(w, b, Xin):
12     return np.dot(Xin, w) + b
13
14 idx = np.random.permutation(1000)
15 TRAIN = idx[0:800]
16 VAL = idx[800:900]
17 TEST = idx[900:]
18
19 ytrain = y[TRAIN]
20 Xtrain = X[TRAIN, :]
21 yval = y[VAL]
22 Xval = X[VAL, :]
23
24 err = np.zeros(50)
25
26 # use cross validation to pick the best features to use
27 for d in range(1,51):
28     w, b = fit(Xtrain[:, 0:d], ytrain)
29     yval_hat = predict(w, b, Xval[:,0:d])
30     err[d-1] = np.mean((yval_hat - yval)**2)
31
32 d_best = np.argmin(err) + 1
33
34 Xtot = np.concatenate((Xtrain, Xval), axis=0)
35 ytot = np.concatenate((ytrain, yval), axis=0)
36
37 w, b = fit(Xtot[:, 0:d_best], ytot)
38
39 ytest = y[TEST]
40 Xtest = X[TEST, :]
41
42 ytot_hat = predict(w, b, Xtot[:, 0:d_best])
43 train_error = np.mean((ytot_hat - ytot) **2)
44 ytest_hat = predict(w, b, Xtest[:, 0:d_best])
45 test_error = np.mean((ytest_hat - ytest) **2)
46
47 print('Train error = ', train_error)
48 print('Test error = ', test_error)
```

Solution:

The answer here is subtle. You may notice that we pick our hyperparameter (d) using our validation set, but then we combine the train and validation sets to get our final w and b . Since the hyperparameter here is just the dimension we are using, the performance of the combined datasets (with d_{best}) should be better than if we just trained on the train set (with d_{best}).

This isn't always the case. Consider picking λ for the amount of regularization.

In general, models get better the more data you give them, but only add the validation set back in if you are

confident the hyperparameter doesn't depend on the number of elements, and that you aren't allowing your model access to the test set.
Since neither of these processes touch the test set, it is still an unbiased estimate of the true error.

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

```
1 # Given dataset of 1000-by-50 feature
2 # matrix X, and 1000-by-1 labels vector
3 import np
4
5
6 def fit(Xin, Yin, lbda):
7     mu = np.mean(Xin, axis=0)
8     Xin = Xin - mu
9     w = np.linalg.solve(np.dot(Xin.T, Xin) + lbda, np.dot(Xin.T, Yin))
10    b = np.mean(Yin) - np.dot(w, mu)
11    return w, b
12
13
14 def predict(w, b, Xin):
15    return np.dot(Xin, w) + b
16
17
18 # Note: X, y are all the train data and label for the entire experiments
19 N_TRAIN = 1000
20 idx = np.random.permutation(N_TRAIN)
21 K_FOLD = 5
22
23 NON_TEST = idx[0: 9 * N_TRAIN // 10]
24 N_PER_FOLD = len(NON_TEST) // K_FOLD
25 TEST = idx[9 * N_TRAIN // 10::]
26
27 # candidates regularization coefficient to choose from
28 lbdas = [0.1, 0.2, 0.3]
29 err = np.zeros(len(lbdas))
30
31 for lbda_idx, lbda in enumerate(lbdas):
32     for i in range(K_FOLD):
33         # CRUCIAL: we use slicing to calculate the index train set and val set should use!
34         # Using the ith fold as the validation set
35         VAL = NON_TEST[i * N_PER_FOLD:(i+1) * N_PER_FOLD]
36         # Using the rest as the train set
37         TRAIN = NON_TEST[:i * N_PER_FOLD] + NON_TEST[(i + 1) * N_PER_FOLD:]
38
39         ytrain = y[TRAIN]
40         Xtrain = X[TRAIN]
41         yval = y[VAL]
42         Xval = X[VAL]
43
44         w, b = fit(Xtrain, ytrain, lbda)
45         yval_hat = predict(w, b, Xval)
46         # accumulate error from this fold of validation set
47         err[lbda_idx] += np.mean((yval_hat - yval)**2)
48
49     # calculate the error for the k-fold validation
50     err[lbda_idx] /= K_FOLD
51
52 lbda_best = lbdas[np.argmin(err)]
```

```
53
54 Xtot = np.concatenate((Xtrain, Xval), axis=0)
55 ytot = np.concatenate((ytrain, yval), axis=0)
56
57 w, b = fit(Xtot, ytot, lbda_best)
58
59 ytest = y[TEST]
60 Xtest = X[TEST]
61
62 ytot_hat = predict(w, b, Xtot)
63 train_error = np.mean((ytot_hat - ytot) ** 2)
64 ytest_hat = predict(w, b, Xtest)
65 test_error = np.mean((ytest_hat - ytest) ** 2)
66
67 print('Best choice of lambda = ', lbda_best)
68 print('Train error = ', train_error)
69 print('Test error = ', test_error)
```
