# Section 05: Subgradients

## 1. K-fold Cross-Validation (Demonstrative code)

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

X = np.random.random((1000,50))
y = np.random.random((1000,))

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 data and labels for the entire experiments
# We first split the data into the training set and test set.
N_SAMPLES = X.shape[0]
idx = np.random.permutation(N_SAMPLES)
K_FOLD = 5

# We use an array of randomized indices to slice the data into the training and test sets.
NON_TEST = idx[0: 9 * N_SAMPLES // 10]
N_PER_FOLD = len(NON_TEST) // K_FOLD
TEST = idx[9 * N_SAMPLES // 10::]

# regularization coefficient candidates 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 indices the training set and validation 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 = np.concatenate((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
```

```
    # After trying all candidates for the regularization coefficient, we select the best lambda.
    lbda_best = lbdas[np.argmin(err)]

    # Fit the model again using all training data from CV.
    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]

    # Predict values using model fit on entire training set and the separate test set, and report error.
    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)
```

## 2. Lasso and CV (Demonstrative Code)

```
import numpy as np

LR = 0.01
NUM_ITERATIONS = 500

# NOTE: here, X and Y represent only the training data, not the overall dataset (train + test).
X = np.random.random((1000, 50))
Y = np.random.random((1000,))

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

def fit(Xin, Yin, l1_penalty) :
   # no_of_training_examples, no_of_features
   m, n = Xin.shape

   # weight initialization
   w = np.zeros(n)
   b = 0

   # gradient descent learning
   for i in range(NUM_ITERATIONS) :
      w, b = update_weights(w, b, Xin, Yin, l1_penalty)

   return w, b

def update_weights(w, b, Xin, Yin, l1_penalty) :
   m, n = Xin.shape
   Y_pred = predict(w, b, Xin)

   # calculate gradients
   dW = np.zeros(n)
   for j in range(n) :
      if w[j] > 0 :
         dW[j] = ( - ( 2 * ( Xin[:, j] ).dot(Yin - Y_pred))
              + l1_penalty ) / m
      else :
         dW[j] = ( - ( 2 * ( Xin[:, j] ).dot(Yin - Y_pred))
              - l1_penalty ) / m

   db = - 2 * np.sum(Yin - Y_pred) / m

   # update weights
   w = w - LR * dW
   b = b - LR * db

   return w, b

def rmse_lasso(w, b, Xin, Yin):
    Y_pred = predict(w, b, Xin)
    return rmse(Yin, Y_pred)

def rmse(a, b):
    return np.sqrt(np.mean(np.square(a - b)))

# candidate values for l1 penalty
l1_penalties = 10 ** np.linspace(-5, -1)
err = np.zeros(len(l1_penalties))

```

```python
# We will perform 10-fold CV. Here, we will create the training and validation sets by
# creating an indices array with randomized index values to use when slicing our training data.
k_fold = 10
num_samples = len(X) // k_fold
indices = np.random.permutation(len(X))

for idx, l1_penalty in enumerate(l1_penalties):
    for k in range(k_fold): #10-fold CV
        # slice larger training set into validation and training sets for each fold
        VAL = indices[k * num_samples : (k + 1) * num_samples]
        TRAIN = np.concatenate((indices[: k * num_samples], indices[(k + 1) * num_samples:]))

        x_train_fold = X[TRAIN]
        y_train_fold = Y[TRAIN]

        x_val_fold = X[VAL]
        y_val_fold = Y[VAL]

        w, b = fit(x_train_fold, y_train_fold, l1_penalty)

        # accumulate error from this fold of validation set
        err[idx] += rmse_lasso(w, b, x_val_fold, y_val_fold)

    #calculate error for kth fold
    err[idx]/=k_fold

l1_penalty_best = l1_penalties[np.argmin(err)]

print('Best choice of l1_penalty = ', l1_penalty_best)
```

# 3. Subgradients

We start with the definition of subgradients before discussing the motivation and its usefulness.

**Definition 1** (subgradients). *A vector $g \in \mathbb{R}^d$ is a subgradient of a convex function $f : D \longrightarrow \mathbb{R}$ at $x \in D \subseteq \mathbb{R}^d$ if*

$$f(y) \geq f(x) + g^T(y - x) \quad \textit{for all } y \in D.$$

One interpretation of subgradient $g$ is that the affine function (of $y$) $f(x) + g^T(y - x)$ is a global underestimator of $f$. Note that if a convex function $f$ is differentiable at $x$ (i.e., $\nabla f(x)$ exists), then $f(y) \geq f(x) + \nabla f(x)^T(y - x)$ is true for all $y \in D$, meaning that $\nabla f(x)$ is a subgradient of $f$ at $x$. But a subgradient can exist even when $f$ is not differentiable at $x$.

(a) Why are subgradients useful in optimization? If $g = 0$ is a subgradient of a function $f$ at $x^*$, what does it imply?

(b) What are the subgradients of $f(x) = \max(x, x^2)$ at 0, with $x \in \mathbb{R}$? (Hint: draw a picture and note that subgradients at a point might not be unique)

(c) Some important results about subgradients are

- If $f$ is convex, then a subgradient of $f$ at $x \in \text{int}(D)$ (interior of the domain of $f$) always exists.
- If $f$ is convex, then $f$ is differentiable at $x$ if and only if $\nabla f(x)$ is the only subgradient of $f$ at $x$.
- A point $x^*$ is a global minimizer of a function $f$ (not necessarily convex) if and only if $g = 0$ is a subgradient of $f$ at $x^*$.