MIT 6.036 Spring 2019: Homework 5

In [4]:
```
import numpy as np
```

Setup

First, download the code distribution for this homework that contains test cases and helper functions.

Run the next code block to download and import the code for this lab.

In [5]:
```
#!rm -rf code_and_data_for_hw05*
#!wget --no-check-certificate --quiet https://introml_oll.odl.mit.edu/6.036/static/homework/hw05/code_and_data_for_hw05.zip
#!unzip code_and_data_for_hw05.zip
#!mv code_and_data_for_hw05/* .

import code_for_hw5 as hw5
```

6) Linear Regression - going downhill

We will now write some general Python code to compute the gradient of the squared-loss objective, following the structure of the expression, and the rules of calculus. Note that this style of writing the gradient functions maps directly into the chain-rule steps required to compute the gradient, but produces code that is inefficient, because of duplicated computations. It is straightforward to implement more efficient versions if you want to use them for larger problems.

6.1) Some basic functions

We start by defining some basic functions for computing the mean squared loss. Note that we want these to work for any value of n. That is, `x` could be a single feature vector or a full data matrix, and similarly for `y`.

In [6]:
```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def lin_reg(x, th, th0):
    """ Returns the predicted y

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 0.]])
    >>> lin_reg(X, th, th0).tolist()
    [[1.05, 2.05, 3.05, 4.05]]
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> lin_reg(X, th, th0).tolist()
    [[3.05, 4.05, 5.05, 6.05]]
    """
    return np.dot(th.T, x) + th0
def square_loss(x, y, th, th0):
    """ Returns the squared loss between y_pred and y

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> square_loss(X, Y, th, th0).tolist()
    [[4.2025, 3.4224999999999985, 5.0625, 3.8025000000000007]]
    """
    return (y - lin_reg(x, th, th0))**2
def mean_square_loss(x, y, th, th0):
    """ Return the mean squared loss between y_pred and y

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> mean_square_loss(X, Y, th, th0).tolist()
    [[4.1225]]
    """
    # the axis=1 and keepdims=True are important when x is a full matrix
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True)
```

6.2) Gradients with respect to θ

Now, let's compute the gradients with respect to θ. Make sure that they work both for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.

In [7]:

```
# Write a function that returns the gradient of lin_reg(x, th, th0)
# with respect to th
def d_lin_reg_th(x, th, th0):
    """ Returns the gradient of lin_reg(x, th, th0) with respect to th

    Note that for array (rather than vector) x, we get a d x n
    result. That is to say, this function produces the gradient for
    each data point i ... n, with respect to each theta, j ... d.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]); th0 = np.array([[ 2.]])
    >>> d_lin_reg_th(X[:,0:1], th, th0).tolist()
    [[1.0], [1.0]]

    >>> d_lin_reg_th(X, th, th0).tolist()
    [[1.0, 2.0, 3.0, 4.0], [1.0, 1.0, 1.0, 1.0]]
    """
    return x

# Write a function that returns the gradient of square_loss(x, y, th, th0) with
# respect to th.  It should be a one-line expression that uses lin_reg and
# d_lin_reg_th.
def d_square_loss_th(x, y, th, th0):
    """Returns the gradient of square_loss(x, y, th, th0) with respect to
       th.

       Note: should be a one-line expression that uses lin_reg and
       d_lin_reg_th (i.e., uses the chain rule).

       Should work with X, Y as vectors, or as arrays. As in the
       discussion of d_lin_reg_th, this should give us back an n x d
       array -- so we know the sensitivity of square loss for each
       data point i ... n, with respect to each element of theta.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist()
    [[4.1], [4.1]]

    >>> d_square_loss_th(X, Y, th, th0).tolist()
    [[4.1, 7.399999999999999, 13.5, 15.600000000000001], [4.1, 3.6999999999999993, 4.5, 3.9000000000000004]]

    """
    dths = d_lin_reg_th(x, th, th0) # d x n
    hs = lin_reg(x, th, th0) - y # 1 x n
    return dths * hs * 2
```

```
# Write a function that returns the gradient of mean_square_loss(x, y, th, th0) with
# respect to th.  It should be a one-line expression that uses d_square_loss_th.
def d_mean_square_loss_th(x, y, th, th0):
    """ Returns the gradient of mean_square_loss(x, y, th, th0) with
        respect to th.

        Note: It should be a one-line expression that uses d_square_loss_th.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_mean_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist()
    [[4.1], [4.1]]

    >>> d_mean_square_loss_th(X, Y, th, th0).tolist()
    [[10.15], [4.05]]
    """
    # print("X =", repr(X))
    # print("Y =", repr(Y))
    # print("th =", repr(th), "th0 =", repr(th0))
    d, n = x.shape
    return np.sum(d_square_loss_th(x, y, th, th0), axis=1, keepdims=True) / n
```

In [8]:
```
def mytest(a, e):
    assert a == e, f"Actual: '{a}' <> Expected: '{e}'"
```

In [9]:
```
X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
th = np.array([[ 1.  ], [ 0.05]]); th0 = np.array([[ 2.]])
print(X[:, 0:1])
mytest(d_lin_reg_th(X[:,0:1], th, th0).tolist(),
[[1.0], [1.0]])

mytest(d_lin_reg_th(X, th, th0).tolist(),
[[1.0, 2.0, 3.0, 4.0], [1.0, 1.0, 1.0, 1.0]])
```

```
[[1.]
 [1.]]
```

In [10]:
```
X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
mytest(
    d_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist(),
    [[4.1], [4.1]]
)
mytest(
    d_square_loss_th(X, Y, th, th0).tolist(),
    [[4.1, 7.399999999999999, 13.5, 15.600000000000001], [4.1, 3.6999999999999993, 4.5, 3.9000000000000004]]
)
```

In [11]:
```
X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
mytest(
    d_mean_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist(),
    [[4.1], [4.1]])

mytest(
    d_mean_square_loss_th(X, Y, th, th0).tolist(),
    [[10.15], [4.05]]
)
```

6.3) Gradients with respect to θ_0

Now, let's compute the gradients with respect to θ_0. Make sure that they work both for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.

In [12]:

```
# Write a function that returns the gradient of lin_reg(x, th, th0)
# with respect to th0. Hint: Think carefully about what the dimensions of the returned value should be!
def d_lin_reg_th0(x, th, th0):
    """ Returns the gradient of lin_reg(x, th, th0) with respect to th0.

    >>> x = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_lin_reg_th0(x, th, th0).tolist()
    [[1.0, 1.0, 1.0, 1.0]]
    """
    d, n = x.shape
    return np.repeat(1.0, n).reshape((1,n))

# Write a function that returns the gradient of square_loss(x, y, th, th0) with
# respect to th0.  It should be a one-line expression that uses lin_reg and
# d_lin_reg_th0.
def d_square_loss_th0(x, y, th, th0):
    """ Returns the gradient of square_loss(x, y, th, th0) with
        respect to th0.

    # Note: uses broadcasting!

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_square_loss_th0(X, Y, th, th0).tolist()
    [[4.1, 3.6999999999999993, 4.5, 3.9000000000000004]]
    """
    return (lin_reg(x, th, th0) - y) * d_lin_reg_th0(x, th, th0) * 2

# Write a function that returns the gradient of mean_square_loss(x, y, th, th0) with
# respect to th0.  It should be a one-line expression that uses d_square_loss_th0.
def d_mean_square_loss_th0(x, y, th, th0):
    """ Returns the gradient of mean_square_loss(x, y, th, th0) with
    respect to th0.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_mean_square_loss_th0(X, Y, th, th0).tolist()
    [[4.05]]
    """
    d, n = x.shape
    return np.sum(d_square_loss_th0(x, y, th, th0), axis=1, keepdims=True) / n
```

7) Going down the ridge

Now, let's add a regularizer. The ridge objective can be implemented as follows:

In [13]:
```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def ridge_obj(x, y, th, th0, lam):
    """ Return the ridge objective value

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> ridge_obj(X, Y, th, th0, 0.0).tolist()
    [[4.1225]]
    >>> ridge_obj(X, Y, th, th0, 0.5).tolist()
    [[4.623749999999999]]
    >>> ridge_obj(X, Y, th, th0, 100.).tolist()
    [[104.37250000000002]]
    """
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True) + lam * np.linalg.norm(th)**2
```

Let's extend our previous code for the gradient of the mean square loss to compute the gradient of the ridge objective w ith respect to θ. Our previous solutions for the non-ridge case: `d_mean_square_loss_th` and `d_mean_square_loss_th0` will be defined for you in the grader, so feel free to call them!

In [14]:

```
def d_ridge_obj_th(x, y, th, th0, lam):
    """Return the derivative of tghe ridge objective value with respect
    to theta.

    Note: uses broadcasting to add d x n to d x 1 array below

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_ridge_obj_th(X, Y, th, th0, 0.0).tolist()
    [[10.15], [4.05]]
    >>> d_ridge_obj_th(X, Y, th, th0, 0.5).tolist()
    [[11.15], [4.1]]
    >>> d_ridge_obj_th(X, Y, th, th0, 100.).tolist()
    [[210.15], [14.05]]
    """
    return d_mean_square_loss_th(x, y, th, th0) + th * 2 * lam

def d_ridge_obj_th0(x, y, th, th0, lam):
    """Return the derivative of tghe ridge objective value with respect
    to theta.

    Note: uses broadcasting to add d x n to d x 1 array below

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_ridge_obj_th0(X, Y, th, th0, 0.0).tolist()
    [[4.05]]
    >>> d_ridge_obj_th0(X, Y, th, th0, 0.5).tolist()
    [[4.05]]
    >>> d_ridge_obj_th0(X, Y, th, th0, 100.).tolist()
    [[4.05]]
    """
    return d_mean_square_loss_th0(x, y, th, th0)
```

8) Stochastic gradient

We will now implement stochastic gradient descent in a general way, similar to what we did with gradient descent (`gd`).

The calling conventions for `sgd` are similar to those of `gd` except that we need to pass in the data and labels for the problem.

(Recall that the *stochastic* part refers to using a randomly selected point and corresponding label from the given dataset to perform an update. Therefore, your objective function for a given step wil l need to take this into account.)

- `X` : a standard data array (d by n)
- `y` : a standard labels row vector (1 by n)
- `J` : a cost function whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector) (in that order), and which returns a scalar.
- `dJ` : a cost function gradient (corresponding to `J`) whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector) (also in that order), and which returns a column vector.
- `w0` : an initial value of weight vector w, which is a column vector.
- `step_size_fn` : a function that is given the (zero-indexed) iteration index (an integer) and returns a step size.
- `max_iter` : the number of iterations to perform

It returns a tuple (like `gd`):

- `w` : the value of the weight vector at the final step
- `fs` : the list of values of J found during all the iterations
- `ws` : the list of values of w found during all the iterations

Note: `w` should be the value one gets after applying stochastic gradient descent to `w0` for `max_iter-1` iterations (we call this the final step). The first element of `fs` should be the value of `J` calculated with `w0` , and `fs` should have length `max_iter` ; similarly, the first element of `ws` should be `w0` , and `ws` should have length `max_iter` .

You might find the function `np.random.randint(n)` useful in your implementation.

Hint: This is a short function; our implementation is around 10 lines.

The main function to implement is below.

In [130…

```
def sgd(X, y, J, dJ, w0, step_size_fn, max_iter):
    """Implements stochastic gradient descent

    Inputs:
    X: a standard data array (d by n)
    y: a standard labels row vector (1 by n)

    J: a cost function whose input is a data point (a column vector),
    a label (1 by 1) and a weight vector w (a column vector) (in that
    order), and which returns a scalar.

    dJ: a cost function gradient (corresponding to J) whose input is a
    data point (a column vector), a label (1 by 1) and a weight vector
    w (a column vector) (also in that order), and which returns a
    column vector.

    w0: an initial value of weight vector www, which is a column
    vector.

    step_size_fn: a function that is given the (zero-indexed)
    iteration index (an integer) and returns a step size.

    max_iter: the number of iterations to perform

    Returns: a tuple (like gd):
    w: the value of the weight vector at the final step
    fs: the list of values of JJJ found during all the iterations
    ws: the list of values of www found during all the iterations

    """
    d, n = X.shape
    ws = [w0]
    fs = [J(X[:,0:1], y[:,0:1], w0)]
    np.random.seed(0)
    for t in range(max_iter-1):
        i = np.random.randint(n)
        step_size = step_size_fn(t)
        xi = X[:,i:i+1]
        yi = y[:,i:i+1]
        ws += [ws[-1] - step_size * dJ(xi, yi, ws[-1])]
        fs += [J(xi, yi, ws[-1])]
    return ws[-1], fs, ws
```

In [128…

```
a = np.array([0.36, 0.028653685126009333])
b = np.array([0.16000000000000003, 0.0008607446103289605])
a / b
```

Out[128]:
```
array([ 2.25      , 33.28941568])
```

The test cases for this problem are provided below (but, as always, you are encouraged to write more if you want to better test your code!). They rely on the function num_grad (taken from the previous week's homework), also provided.

In [124…
```
def rv(value_list):
    return np.array([value_list])

def cv(value_list):
    return np.transpose(rv(value_list))
"""
def f1(x):
    return float((2 * x + 3)**2)

def df1(x):
    return 2 * 2 * (2 * x + 3)

def f2(v):
    x = float(v[0]); y = float(v[1])
    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2

def df2(v):
    x = float(v[0]); y = float(v[1])
    return cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
              2 * (-1. + x + y),
              2 * (-1. + x + y)])
"""
```

Out[124]:
```
'\ndef f1(x):\n    return float((2 * x + 3)**2)\n\ndef df1(x):\n    return 2 * 2 * (2 * x + 3)\n\ndef f2(v):\n    x = float(v
[0]); y = float(v[1])\n    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2\n\ndef df2(v):\n    x = float(v
[0]); y = float(v[1])\n    return cv([(-3. + x) * (-2. + x) * (1. + x) +               (-3. + x) * (-2. + x) * (3. + x) +
(-3. + x) * (1. + x) * (3. + x) +               (-2. + x) * (1. + x) * (3. + x) +               2 * (-1. + x + y),\n
2 * (-1. + x + y)])\n'
```

In [125…

```
def num_grad(f):
    def df(x):
        g = np.zeros(x.shape)
        delta = 0.001
        for i in range(x.shape[0]):
            xi = x[i,0]
            x[i,0] = xi - delta
            xm = f(x)
            x[i,0] = xi + delta
            xp = f(x)
            x[i,0] = xi
            g[i,0] = (xp - xm)/(2*delta)
        return g
    return df

def downwards_line():
    X = np.array([[0.0, 0.1, 0.2, 0.3, 0.42, 0.52, 0.72, 0.78, 0.84, 1.0],
                  [1.0, 1.0, 1.0, 1.0, 1.0,  1.0,  1.0,  1.0,  1.0,  1.0]])
    y = np.array([[0.4, 0.6, 1.2, 0.1, 0.22, -0.6, -1.5, -0.5, -0.5, 0.0]])
    return X, y

X, y = downwards_line()

def J(Xi, yi, w):
    # translate from (1-augmented X, y, theta) to (separated X, y, th, th0) format
    return float(ridge_obj(Xi[:-1,:], yi, w[:-1,:], w[-1:,:], 0))

def dJ(Xi, yi, w):
    def f(w): return J(Xi, yi, w)
    return num_grad(f)(w)
```

In [126…

```
ans=sgd(X, y, J, dJ, cv([0., 0.]), lambda i: 0.1, 1000)
# print(ans[0])
# print(ans[1][0])
# print(ans[1][-1])
# print(ans[2][0])
# print(ans[2][-1])
```

```
[-1.20232666]
```

9) Predicting mpg values

We will now try to synthesize the functions we have written in order to perform ridge regression on the auto-mpg dataset from lab03. Unlike in lab03, we will now try to predict the actual mpg values of the cars, instead of whether they are above or below the median mpg!

As a reminder, the dataset is as follows:

```
1. mpg:           continuous
2. cylinders:     multi-valued discrete
3. displacement:  continuous
4. horsepower:    continuous
5. weight:        continuous
6. acceleration:  continuous
7. model year:    multi-valued discrete
8. origin:        multi-valued discrete
9. car name:      string (many values)
```

For convenience, we will choose to not include `model year` and `car name` as features. For the remaining features, we again have the option to keep the raw values, standardize them, or use a one-hot encoding.

With this considered, we decide to standardize or one-hot encode all features in this section (we encourage you, though, to try raw features on your own time to see how their performance matches your expectations!).

One additional step we perform is to standardize the output values. Note that we did not have to worry about this in a classification context, as all outputs were ± 1. In a regression context, standardizing the output values can have practical performance gains, again due to better numerical performance of learning algorithms on data which is smaller in magnitude.

The metric we will use to measure the quality of our learned predictors is **Root Mean Square Error (RMSE).** RMSE is defined as follows:

$$ \text{RMSE} = \sqrt{ \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - f(x^{(i)}) \right)^2 } $$
where f is our learned predictor: in this case, $f(x) = \theta \cdot x + \theta_0$. This gives a measure of how far away the true values are from the predicted values, measured in units of mpg.

Note: One very important thing to keep in mind when employing standardization is that we need to reverse the standardization when we want to report results. If we standardize output values in the training set by subtracting μ and dividing by σ, we need to take care to:

1. Perform standardization with the same values of μ and σ on the test set (Why?) before predicting outputs using our learned predictor.
2. Multiply the RMSE calculated on the test set by a factor of σ to report test error (Why?)

Given all of this, we now will try using:

- Two choices of feature set:
    1. `[cylinders=standard, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`
    2. `[cylinders=one_hot, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`
- Polynomial features (we will construct the polynomial features after having standardized the input data) of orders 1-3
- Different choices of the regularization parameter, λ. Although, ideally, you would run a grid search over a large range of λ, we will ask you to look at the choices $\lambda = \{0.01, 0.02, \cdots, 0.1\}$ for polynomial features of orders 1 and 2, and the choices $\lambda = \{20, 40, \cdots, 200\}$ for polynomial features of order 3 (as this is approximately where we found the optimal λ to lie).

We will use 10-fold cross-validation to try all possible combinations of these feature choices and test which is best.

Your functions written above will be called by `ridge_min`, (defined for you below), which takes a dataset (X, y) and a hyperparameter, λ as input and returns θ and θ_0 minimizing the ridge regression objective using SGD (this is the analogue of the `svm_min` function that you wrote for homework last week). The learning rate and number of iterations are fixed in this function, and should not be modified for the purpose of answering the below questions (although you should feel free to experiment with these if you are interested!) This function will then further be called by `xval_learning_alg` (also defined below), which returns the average RMSE across all (here, 10) splits of your data when performing cross-validation.

Note: Even though these functions are also contained in the code file being imported (`code_for_hw5.py`), you should run the below code block so that they will use the version of the functions you have written above, and not the blank versions in the code file.

In [64]:
```
#Concatenates the gradients with respect to theta and theta_0
def ridge_obj_grad(x, y, th, th0, lam):
    grad_th = d_ridge_obj_th(x, y, th, th0, lam)
    grad_th0 = d_ridge_obj_th0(x, y, th, th0, lam)
    return np.vstack([grad_th, grad_th0])

def ridge_min(X, y, lam):
    """ Returns th, th0 that minimize the ridge regression objective

    Assumes that X is NOT 1-extended. Interfaces to our sgd by 1-extending
    and building corresponding initial weights.
    """
    def svm_min_step_size_fn(i):
        return 0.01/(i+1)**0.5

    d, n = X.shape
    X_extend = np.vstack([X, np.ones((1, n))])
```

```
    w_init = np.zeros((d+1, 1))

    def J(Xj, yj, th):
        return float(ridge_obj(Xj[:-1,:], yj, th[:-1,:], th[-1:,:], lam))

    def dJ(Xj, yj, th):
        return ridge_obj_grad(Xj[:-1,:], yj, th[:-1,:], th[-1:,:], lam)

    np.random.seed(0)
    w, fs, ws = sgd(X_extend, y, J, dJ, w_init, svm_min_step_size_fn, 1000)
    return w[:-1,:], w[-1:,:]

#First finds a predictor on X_train and X_test using the specified value of lam
#Then runs on X_test, Y_test to find the RMSE
def eval_predictor(X_train, Y_train, X_test, Y_test, lam):
    th, th0 = ridge_min(X_train, Y_train, lam)
    return np.sqrt(mean_square_loss(X_test, Y_test, th, th0))

#Returns the mean RMSE from cross validation given a dataset (X, y), a value of lam,
#and number of folds, k
def xval_learning_alg(X, y, lam, k):
    _, n = X.shape
    idx = list(range(n))
    np.random.seed(0)
    np.random.shuffle(idx)
    X, y = X[:,idx], y[:,idx]

    split_X = np.array_split(X, k, axis=1)
    split_y = np.array_split(y, k, axis=1)

    score_sum = 0
    for i in range(k):
        X_train = np.concatenate(split_X[:i] + split_X[i+1:], axis=1)
        y_train = np.concatenate(split_y[:i] + split_y[i+1:], axis=1)
        X_test = np.array(split_X[i])
        y_test = np.array(split_y[i])
        score_sum += eval_predictor(X_train, y_train, X_test, y_test, lam)
    return score_sum/k
```

In [ ]:

```
# Returns a list of dictionaries.  Keys are the column names, including mpg.
auto_data_all = hw5.load_auto_data('auto-mpg-regression.tsv')

# The choice of feature processing for each feature, mpg is always raw and
# does not need to be specified.  Other choices are hw5.standard and hw5.one_hot.
# 'name' is not numeric and would need a different encoding.
features1 = [('cylinders', hw5.standard),
             ('displacement', hw5.standard),
             ('horsepower', hw5.standard),
             ('weight', hw5.standard),
             ('acceleration', hw5.standard),
             ('origin', hw5.one_hot)]

features2 = [('cylinders', hw5.one_hot),
             ('displacement', hw5.standard),
             ('horsepower', hw5.standard),
             ('weight', hw5.standard),
             ('acceleration', hw5.standard),
             ('origin', hw5.one_hot)]

# Construct the standard data and label arrays
#auto_data[0] has the features for choice features1
#auto_data[1] has the features for choice features2
#The labels for both are the same, and are in auto_values
auto_data = [0, 0]
auto_values = 0
auto_data[0], auto_values = hw5.auto_data_and_values(auto_data_all, features1)
auto_data[1], _ = hw5.auto_data_and_values(auto_data_all, features2)

#standardize the y-values
auto_values, mu, sigma = hw5.std_y(auto_values)

#-------------------------------------------------------------------------------
# Analyze auto data
#-------------------------------------------------------------------------------

#Your code for cross-validation goes here
#Make sure to scale the RMSE values returned by xval_learning_alg by sigma,
#as mentioned in the lab, in order to get accurate RMSE values on the dataset


res = []
for feature_set_i in (0, 1):
    for order, lam_set in (
        (1, (np.arange(11))*0.01),
        (2, (np.arange(11))*0.01),
        (3, (np.arange(0,220,20)))
    ):
```

```
        print('feature_i:', feature_set_i)
        print('order:', order)
        print('lam_set:', lam_set)

        data = hw5.make_polynomial_feature_fun(order)(auto_data[feature_set_i])
        for lam in lam_set:
            print('lam', lam)
            mean_rmse = xval_learning_alg(data, auto_values, lam, 10) * sigma
            print(mean_rmse)
            res += [(mean_rmse, lam, order, feature_set_i)]
print(sorted(res)[:5])
```

```
feature_i: 0
order: 1
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[4.27349492]]
lam 0.01
[[4.27439892]]
lam 0.02
[[4.27535367]]
lam 0.03
[[4.27635832]]
lam 0.04
[[4.27741202]]
lam 0.05
[[4.27851395]]
lam 0.06
[[4.27966327]]
lam 0.07
[[4.28085918]]
lam 0.08
[[4.28210088]]
lam 0.09
[[4.28338758]]
lam 0.1
[[4.28471851]]
feature_i: 0
order: 2
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[4.02634119]]
lam 0.01
[[4.02714651]]
lam 0.02
[[4.02801017]]
lam 0.03
[[4.0289311]]
lam 0.04
```

```
[[4.02990823]]
lam 0.05
[[4.03094054]]
lam 0.06
[[4.03202697]]
lam 0.07
[[4.03316652]]
lam 0.08
[[4.03435819]]
lam 0.09
[[4.035601]]
lam 0.1
[[4.03689397]]
feature_i: 0
order: 3
lam_set: [  0  20  40  60  80 100 120 140 160 180 200]
lam 0
[[91158721.07809679]]
lam 20
[[6.47860272]]
lam 40
[[6.02418287]]
lam 60
[[6.03936845]]
lam 80
[[6.03256196]]
lam 100
[[6.03126881]]
lam 120
[[6.04675354]]
lam 140
[[6.07874785]]
lam 160
[[6.12733435]]
lam 180
[[6.19787407]]
lam 200
[[6.27648811]]
feature_i: 1
order: 1
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[4.14164798]]
lam 0.01
[[4.1431706]]
lam 0.02
[[4.1447585]]
lam 0.03
```

```
[[4.14641043]]
lam 0.04
[[4.14812515]]
lam 0.05
[[4.14990145]]
lam 0.06
[[4.15173813]]
lam 0.07
[[4.153634]]
lam 0.08
[[4.15558791]]
lam 0.09
[[4.15759869]]
lam 0.1
[[4.15966523]]
feature_i: 1
order: 2
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[3.88436985]]
lam 0.01
[[3.88509423]]
lam 0.02
[[3.88586872]]
lam 0.03
[[3.88669241]]
lam 0.04
[[3.88756441]]
lam 0.05
[[3.88848386]]
lam 0.06
[[3.88944989]]
lam 0.07
[[3.89046164]]
lam 0.08
[[3.89151829]]
lam 0.09
[[3.89261901]]
lam 0.1
[[3.89376299]]
feature_i: 1
order: 3
lam_set: [  0  20  40  60  80 100 120 140 160 180 200]
lam 0
[[3741589.4937768]]
lam 20
[[5.73658168]]
lam 40
```

```
[[5.91137479]]
lam 60
[[5.99942908]]
lam 80
[[6.04674106]]
lam 100
[[6.08945111]]
lam 120
[[6.1372991]]
lam 140
[[6.19289135]]
lam 160
[[6.25670678]]
lam 180
[[6.32933722]]
lam 200
[[6.41186557]]
[(array([[3.88436985]]), 0.0, 2, 1), (array([[3.88509423]]), 0.01, 2, 1), (array([[3.88586872]]), 0.02, 2, 1), (array([[3.88669
241]]), 0.03, 2, 1), (array([[3.88756441]]), 0.04, 2, 1)]
```