You may find the lecture notes on regression helpful as you do this homework.

1) Intro to linear regression

So far, we have been looking at classification, where predictors are of the form

$$\hat{y}^{(i)} = sign(\theta^T x^{(i)} + \theta_0)$$

where $\hat{y}^{(i)}$ is our prediction of the corresponding label $y^{(i)}$ making a binary classification as to whether example $x^{(i)}$ belongs to the positive or negative class of examples.

In many problems, we want to predict a real value, such as the actual gas mileage of a car, or the concentration of some chemical. Luckily, we can use much of the mechanism we have already developed, and make predictors of the form:

$$\hat{y}^{(i)} = \theta^T x^{(i)} + \theta_0 \quad .$$

This is called a *linear regression* model.

We would like to learn a linear regression model from examples. Assume X is a d by n array (as before) but that Y is a 1 by n array of floating-point numbers (rather than +1 or -1). Given data (X, Y) we need to find θ, θ_0 that does a good job of making predictions on new data drawn from the same source.

We will approach this problem by formulating an objective function. There are many possible reasonable objective functions that implicitly make slightly different assumptions about the data, but they all typically have the form:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0)$$

For regression, we most frequently use *squared loss*, in which

$$L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) = (\hat{y}^{(i)} - y^{(i)})^2 = (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2 \ .$$

We might start by simply trying to minimize the average squared loss on the training data; this is called the *empirical risk* or *mean square error*:

$$J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) \ .$$

Later, we will add in a regularization term.

We will see later in this assignment that we can find a closed form matrix formula (requiring a matrix inverse) for the optimal θ in a linear regression formula. Being able to solve a machine-learning problem in closed form is very awesome! But inverting a matrix is computationally expensive (a bit less than $O(m^3)$ where m is the dimension of our matrix), and so, as our data sets get larger, we will need to find some more efficient or approximate ways to approach the problem.

For your convenience, we have copied the hands-on section into a colab notebook, which may be found here. You can alternatively fill in these functions in `code_for_hw5.py`, which is part of this set of files (the other files will be useful for the last part of the homework).

Let's start by thinking about gradient descent to attack this problem:

1A) What is the gradient of the empirical risk with respect to θ? We can see that it is of the form:

$$\nabla_\theta J_{emp}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} g(x^{(i)}, y^{(i)})$$

where $x^{(i)}, y^{(i)}$ are the i^{th} data point and its label.

Write an expression for $g(x^{(i)}, y^{(i)})$ using the symbols: `x_i`, `y_i`, `theta` and `theta_0`, where $g(x^{(i)}, y^{(i)})$ is the derivative of the L_s function described above with respect to θ. Remember that you can use `@` for matrix product, and you can use `transpose(v)` to transpose a vector. Note that this $g(\cdot)$ function is *just* the derivative with respect to a single data point $x^{(i)}, y^{(i)}$. We'll build up to the gradient of J_{emp} in a moment.

$g(x^{(i)}, y^{(i)}) =$ 2 * (transpose(theta)@x_i +theta_0 - y_i) * x_i

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

1B) What is the gradient of the empirical risk **now with respect to** θ_0? We can see that it is of a similar form:

$$\nabla_{\theta_0} J_{emp}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} g_0(x^{(i)}, y^{(i)}) \ .$$

Write an expression for $g_0(x^{(i)}, y^{(i)})$ using the symbols: `x_i`, `y_i`, `theta` and `theta_0`.

$g_0(x^{(i)}, y^{(i)}) =$ 2 * (transpose(theta)@x_i +theta_0 - y_i)

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

1C) Next we're interested in the gradient of the empirical loss with respect to θ, $\nabla_\theta J_{emp}$, but now for a whole data set X (of dimensions d by n).

Write an expression for $\nabla_\theta J_{emp}$ using the symbols: `X` and `Y` for the full set of data, `theta`, `theta_0`, and `n`. Here `X` has dimensions d by n, and `Y` has dimensions 1 by n. Remember that you can use `@` for matrix product, and you can use `transpose(a)` to transpose a vector or array.

$\nabla_\theta J_{emp} =$ 2/n *X @(transpose(X) @ theta + theta_0 - transpose(Y))

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2) Sources of Error

Recall that *structural* error arises when the hypothesis class cannot represent a hypothesis that performs well on the test data and *estimation* error arises when the parameters of a hypothesis cannot be estimated well based on the training data. (You can also refer to here in the notes.)

Following is a collection of potential cures for a situation in which your learning algorithm generates a hypothesis with a high test error.

For each one, indicate whether it can **can reduce** structural error, estimation error, both, or neither.

2A) Penalize $||\theta||^2$ during training.

Can reduce:

- ○ structural error
- ◉ estimation error
- ○ both
- ○ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2B) Penalize $||\theta||^2$ during testing.

Can reduce:

- ○ structural error
- ○ estimation error
- ○ both
- ◉ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2C) Increase the amount of training data.

Can reduce:

- ○ structural error
- ◉ estimation error
- ○ both
- ○ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2D) Increase the order of a fixed polynomial basis.

Can reduce:

- (•) structural error
- ( ) estimation error
- ( ) both
- ( ) neither

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

2E) Decrease the order of a fixed polynomial basis.

Can reduce:

- ( ) structural error
- (•) estimation error
- ( ) both
- ( ) neither

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

3) Minimizing empirical risk

We can also solve regression problems analytically.

Remember the definition of *squared loss*,

$$L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) = (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2$$

and *empirical risk*:

$$J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0)$$

Later, we will add in a regularization term.

For simplicity in this section, assume that we are handling the constant term, θ_0, by adding a dimension to the input feature vector that always has the value 1. To review why this works, take a look at the introduction to problem 1 in HW2.

Let data matrix $Z = X^T$ be n by d, let target output vector $T = Y^T$ be n by 1, and recall that θ is d by 1. Then we can write the whole linear regression prediction as $Z\theta$.

3A) T is the n by 1 vector of target output values. Write an equation expressing the mean squared loss of θ in terms of Z, T, n, and θ. Hint: note that this loss $J(\theta)$ is a scalar, a sum of squared terms divided by n; we can write it as $(W^T W)/n$ for a column vector W.

Enter your answer as a Python expression. You can use symbols `Z`, `T`, `n` and `theta`. Recall that our expression syntax includes `transpose(x)` for transpose of an array, `inverse(x)` for the inverse of an array, and `x@y` to indicate a matrix product of two arrays.

$J(\theta) =$ transpose(Z@theta - T)@(Z@theta - T)/n

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

Now, how can we find the minimizing θ, given Z and T? Take the gradient (yes, even with a matrix expression), set it to zero(s) and solve for θ.

3B) What is $\nabla_\theta J(\theta)$ in terms of Z, T, θ, and n? You can use matrix derivatives or, compute the answer for some individual elements and deduce the matrix form.

$\nabla_\theta J(\theta) =$ 2/n *transpose(Z) @ (Z@theta- T)

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3C) What if you set this equation to 0 and solve for θ^*, the optimal θ? Hint: It's ok to ignore the constant scaling factor.

$\theta^* =$

- ○ $(Z^T T)^{-1}(Z^T Z)$
- ◉ $(Z^T Z)^{-1} Z^T T$
- ○ $(Z Z^T)^{-1} Z T^T$

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: $(Z^T Z)^{-1} Z^T T$

Explanation:

We solve for θ in $\frac{2}{n} \cdot Z^T(Z\theta - T) = 0$. We first expand this to obtain $\left(Z^T Z\right)\theta - Z^T T = 0$. This can then be rewritten $\left(Z^T Z\right)\theta = Z^T T$. This is then a system of linear equations that can be solved as $\theta = (Z^T Z)^{-1} Z^T T$, as desired.

3D) Just converting back to the data matrix format we have been using (not transposed), we have

$\theta^* =$

- ○ $(XY^T)^{-1}(XX^T)$
- ○ $(X^TX)^{-1}X^TY$
- ◉ $(XX^T)^{-1}XY^T$

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3E) Now implement θ^* as found in **3D)**, using symbols `X` and `Y` for the data matrix and outputs, and `np.dot`, `np.transpose`, `np.linalg.inv`.

```
1 # Enter an expression to compute and set th to the optimal theta
2 th = np.linalg.inv(X@X.T) @ X @ Y.T
```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

4) Adding regularization

Although we don't have the same notion of margin maximization as with the SVM formulation for classification, there is still a good reason to *regularize* or put pressure on the coefficient vector θ to prevent the model from fitting the training data too closely, especially in cases where we have few data points and many features.

And, as it happens, this same regularization will help address a problem that you might have anticipated when finding the analytical solution for θ, which is that XX^T might not be invertible (where we are using the definition of X as in problem 3, where each column of X is a d-length vector representing a d-feature sample point and there are n columns in X (or equivalently, there are n training examples)).

Consider this matrix:

```
X = np.array([[1, 2], [2, 3], [3, 5], [1, 4]])
```

Is XX^T invertible? If not, what's the problem? Mark all that are true.

- [ ] It is invertible
- [ ] It is not invertible because X is not square
- [ ] It is not invertible because two columns of X are linearly dependent
- [x] It is not invertible because the rows of XX^T are linearly dependent
- [x] It is not invertible because n is smaller than d
- [ ] We cannot compute the transpose of X

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution:

✗ It is invertible

✗ It is not invertible because X is not square

✗ It is not invertible because two columns of X are linearly dependent

✓ It is not invertible because the rows of XX^T are linearly dependent

✓ It is not invertible because n is smaller than d

✗ We cannot compute the transpose of X

Explanation:

We can multiply out XX^T to obtain

$$XX^T = \begin{bmatrix} 5 & 8 & 13 & 9 \\ 8 & 13 & 21 & 14 \\ 13 & 21 & 34 & 23 \\ 9 & 14 & 23 & 17 \end{bmatrix}$$

Immediately, we notice that rows 1 and 2 add to row 3, so the rows of X are not linearly independent (4 is true). Therefore, we conclude that XX^T is not full rank, and thus not invertible (1 is false).

For generic X, XX^T may be invertible, even if X is not square; simply consider $X' = X^T$, where $X'X'^T$ is a full rank 2 by 2 matrix (2 is false). Similarly, XX^T may be invertible even if the columns of X are linearly dependent; consider matrix

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

We see that AA^T is an invertible 2 by 2 matrix, even though columns 1 and 3 are the same (3 is false).

If n is smaller than d, then X has maximum rank n, and so XX^T has maximum rank n. Since XX^T is a d by d matrix, where $d > n$, XX^T is not invertible (5 is true).

Finally, we can calculate X^T for any X, regardless of its rank (6 is false).

5) Evaluation

We have been hired by Buy 'n' Large to deliver a predictor of change in sales volume from last year, for each of their stores. We have a machine-learning algorithm that can be used with regularization parameter λ. Our overall objective is to deliver a predictor that minimizes squared loss on predictions when actually used by the company. We have three data sets: D_{train}, D_{test} and D_{real}, each of size n. The D_{real} is owned by the company.

We will focus on a linear predictor with parameters θ without the offset parameter θ_0 for simplicity and use regularizer $\lambda\|\theta\|^2$, where λ is the regularization parameter. There are several phases of the training process (as represented in problems 5A through 5D below), and we need to select the appropriate objective for each of those tasks. In the problems below, we will have different expressions with different "slots" (A, B, C, D) to fill from, picking among the following options available to us related to

- what the minimization is over,
- the dataset used,
- the predictor used, and
- whether regularization is added.

Fill in the slots (A,B,C,D) for each phase by choosing the expressions for the indicated slots. The available expressions are shown below; please enter the index for the expression for each of the slots.

1. 0
2. θ
3. $\theta_{best}(\lambda)$,
4. θ^*,
5. λ,
6. λ^*,
7. $\lambda\|\theta\|^2$,
8. $\lambda^*\|\theta\|^2$,
9. D_{train},
10. D_{test},
11. $D_{train} \cup D_{test}$,
12. D_{real}

Note that $\theta_{best}(\lambda)$ is a value of θ that is a function of λ; θ^*, λ^* are specific values found as described below; θ and λ are variables that range over d-dimensional column vectors and positive reals, respectively.

5A) Selecting the best hypothesis (parameters θ) for some fixed value of the regularization parameter λ. Call this $\theta_{best}(\lambda)$.

$$\theta_{best}(\lambda) = \arg\min_A \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 4 indices for [A, B, C, D]: [2, 9, 2, 7]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

5B) Selecting the best value of the regularization parameter λ. We will call this best value λ^*.

$$\lambda^* = \arg\min_A \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 4 indices for [A, B, C, D]: [5, 10, 3, 1]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

5C) Selecting the hypothesis (parameters θ) to deliver to the company. Call this θ^*.

$$\theta^* = \arg\min_A \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 4 indices for [A, B, C, D]: [2, 9, 2, 8]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

5D) Evaluating the actual on-the-job performance ϵ^* of the selected hypothesis θ^*.

$$\epsilon^* = \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 3 indices for [B, C, D]: [12, 4, 1]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

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.

Reminder: For your convenience, we have copied the hands-on section into a colab notebook, which may be found here.. Alternatively, you can work with these functions on your own computer in `code_for_hw5.py`, contained in this zip file. That file has somewhat longer docstrings and doctests for many of these functions and other basic utiliities, that may be useful to you in debugging your implementations. (The other files therein will be useful in the last part of this homework).

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 (of dimension d by 1) or a full data matrix (of dimension d by n), and similarly for `y`.

```
# 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):
    return np.dot(th.T, x) + th0

def square_loss(x, y, th, th0):
    return (y - lin_reg(x, th, th0))**2

def mean_square_loss(x, y, th, th0):
    # 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)
```

These functions will already be defined when you are answering the questions below.

Warm up:

6A)

If X is d by n and Y is 1 by n, what is the dimension of θ? d by 1

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

6B)

If X is d by n and Y is 1 by n, what is the dimension of $\nabla_\theta J_{emp}(\theta, \theta_0)$? d by 1

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

6C) Now let's compute the gradients with respect to θ, make sure that they work 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 the code below, the following values are used in the test cases:

```
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],[0.05]])
th0 = np.array([[0.]])
```

```
# 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):
    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):
    return 2* (y - lin_reg(x, th, th0)) * - d_lin_reg_th(x, th, th0)

# 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):
    return np.mean(d_square_loss_th(x, y, th, th0), axis = 1, keepdims = True)

```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

6D) Now let's compute the gradients with respect to θ_0, make sure that they work for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently. The test cases will include example variables for `X`, `Y`, `th`, and `th0` from 6C above.

```python
# 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
def d_lin_reg_th0(x, th, th0):
    d, n = x.shape
    return np.ones((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):
    return 2 * (y - lin_reg(x, th, th0)) * - d_lin_reg_th0(x, th, th0)

# 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):
    return np.mean(d_square_loss_th0(x, y, th, th0), axis = 1, keepdims = True)

```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

7) Going down the ridge

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

```
# 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 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 with respect to θ. Our previous solutions for the non-ridge case: `d_mean_square_loss_th` and `d_mean_square_loss_th0` are defined for you and you can call them. The test cases will include example variables for `X`, `Y`, `th`, and `th0` from 6C above.

```
def d_ridge_obj_th(x, y, th, th0, lam):
    return d_mean_square_loss_th(x, y, th, th0) + 2 * lam * th

def d_ridge_obj_th0(x, y, th, th0, lam):
    return d_mean_square_loss_th0(x, y, th, th0)

```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

8) Stochastic gradient

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

`sgd` takes the following as input: (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 will 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 test cases are:

```
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)
```

where `num_grad` is taken from homework from the previous week:

```
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
```

```
        Xi, yi = get_rnd_xiyi(X, y)
        #append current data points to lists
        w_list.append(w)
        J_list.append(J(Xi, yi, w))
        #apply the gradient on this data point
        grad = dJ(Xi, yi, w)
        #apply the step
        w = w - step_size_fn(iter) * grad
        if iter == max_iter-2:
            #if this is the last iter, we want to add the
            #w and j at this point without calculating a new w
            #randomly select a data point
            Xi, yi = get_rnd_xiyi(X, y)
            #append current data points to lists
            w_list.append(w)
```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

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.

9A) What is true about leaving features as raw versus deciding to standardize them, in the context of linear regression without regularization?

- [ ] The set of all possible linear regression models learnable on standardized features is smaller than the set of linear regression models learnable on raw features
- [x] The theoretical minimum value of the loss function is the same both with raw and standardized features
- [x] SGD will typically perform better on standardized features than on raw features.

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution:

✗ The set of all possible linear regression models learnable on standardized features is smaller than the set of linear regression models learnable on raw features

✓ The theoretical minimum value of the loss function is the same both with raw and standardized features

✓ SGD will typically perform better on standardized features than on raw features.

Explanation:

1. The set of linear regression models is the same, in both cases. Say that θ and θ_0 are the parameters of a predictor operating on standardized features. Then, the prediction for the example x will be $\hat{y} = \theta \cdot \left(\frac{x-\mu}{\sigma}\right) + \theta_0$, where μ and σ represent the standardization. Note that this is the same as $\hat{y} = \tilde{\theta} \cdot x + \tilde{\theta}_0$, where $\tilde{\theta} = \frac{\theta}{\sigma}$ and $\tilde{\theta}_0 = \theta_0 - \frac{\theta \cdot \mu}{\sigma}$. For any fixed μ, σ, we can vary θ and θ_0 to make $\tilde{\theta}$ and $\tilde{\theta}_0$ equal to whatever values we want. Thus, we can represent any linear predictor with an appropriate choice of parameters in a linear predictor operating on standardized features.

2. From the explanation for 1, the set of possible models is the same with and without standardization. Thus, the predictor minimizing the error in both cases will be the same, causing the theoretical minimum of the loss function to also be the same.

3. At every iteration, SGD takes a step in the direction opposite to the (stochastic) gradient, with the same step size in each "direction" of parameter space. However, with raw features, you would naturally want larger step sizes to update parameters which have a larger range, and smaller ones for those which have a small range. Using a large step size can then often lead to divergence of SGD due to the parameters with a smaller range; using a small step size can lead to very slow convergence due to those parameters with a larger range. Standardizing features largely removes this problem, making it "more acceptable" to use a constant step size across all directions.

9B) What is true about leaving features as raw versus deciding to standardize them, in the context of ridge regression (i.e., we have a nonzero regularizer)?

- [ ] The set of all possible models learnable on standardized features is smaller than the set of models learnable on raw features
- [ ] The theoretical minimum value of the loss function is the same both with raw and standardized features
- [x] SGD will typically perform better on standardized features than on raw features.

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution:

✗ The set of all possible models learnable on standardized features is smaller than the set of models learnable on raw features

✗ The theoretical minimum value of the loss function is the same both with raw and standardized features

✓ SGD will typically perform better on standardized features than on raw features.

Explanation:

1. The set of learnable models is the set of all linear models, as in the previous question. Thus, the answer is the same.

2. While the set of learnable models is the same, for raw features, the regularizer penalizes certain feature weights much more than others. For instance, if the magnitude of a particular feature is very small in the raw data, the learned weight in the absence of a regularizer might be very large. When adding regularization, this feature weight will be driven to be closer to 0, so this feature is disproportionately affected by the regularizer. The optimal weights for both objectives will not then generically correspond to the same model, and the loss values will be different in both cases.

3. Same reason as in 9A.

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 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 that is in a good magnitude range.

The metric we will use to measure the quality of our learned predictors is **Root Mean Square Error (RMSE).** This is useful metric because it gives a sense of the deviation in the nature units of the predictor. 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; we are interested in this value, 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 $\lambda = \{0.0, 0.01, 0.02, \cdots, 0.1\}$ for polynomial features of orders 1 and 2, and the choices $\lambda = \{0, 20, 40, \cdots, 200\}$ 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. We have attached a code file with some predefined methods that will be useful to you here. Alternatively, a google colab link may be found here. If you choose to use the code file, a more detailed description of the roles of the files is below:

The file `code_for_hw5.py` contains functions, some of which will need to be filled in with your definitions from this homework. Your functions are then called by `ridge_min`, defined for you, 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 for you in the same file), which returns the average RMSE across all (here, 10) splits of your data when performing cross-validation. (Note that this RMSE is reported in standardized `y` units; to convert this to RMSE in mpg (miles per gallon), you should multiply this by the `sigma` returned by the `hw5.std_y` function call.)

The file `auto.py` will be used to implement the auto data regression. The file contains code for creating the two feature sets that you are asked to work with here. Transforming those features further with `make_polynomial_feature_fun`, and running the cross-validation function, which uses your implementations in `code_for_hw5.py` (both from `code_for_hw5.py`), you should be able to answer the following questions:

9C) What combination minimizes the average cross-validation RMSE?

Enter a tuple of three numbers (feature_set, polynomial_order, lambda):

(2, 2, 0)

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `(2, 2, 0.0)`

Explanation:

(2, 2, 0.0) gave us the lowest RMSE. We found that λ had a relatively modest impact on RMSE, with small (about zero) values of λ giving the best results, for feature set 2 with 2nd order polynomials.

9D) What is the cross-validation RMSE value (in mpg) that you obtain using the best combination?

Enter an accuracy value: 3.884

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

9E) Say that we really wanted to fit an order 3 polynomial model using the first feature set. What value of lambda minimizes the average cross-validation RMSE, and what is the RMSE value at that value of lambda?

Enter a python list of two numbers [lambda, RMSE_value]: [40, 6.02]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.