# 3

## 3.1

The completed code for `L2NormPenaltyNode` follows:

```
class L2NormPenaltyNode(object):
    """ Node computing l2_reg * ||w||^2 for scalars l2_reg and vector w"""
    def __init__(self, l2_reg, w, node_name):
        """
        Parameters:
        l2_reg: a scalar value >=0 (not a node)
        w: a node for which w.out is a numpy vector
        node_name: node's name (a string)
        """
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.l2_reg = np.array(l2_reg)
        self.w = w

        ## TODO
    def forward(self):
        #print(self.l2_reg, np.dot(self.w.out, self.w.out))
        self.out = self.l2_reg *np.dot(self.w.out, self.w.out)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_w = 2* self.l2_reg * (self.d_out * self.w.out)
        self.w.d_out += d_w
        return self.d_out

    def get_predecessors(self):
        return [self.w]
```

### 3.2

The completed code for `SumNode` follows:

```
class SumNode(object):
    """ Node computing a + b, for numpy arrays a and b"""
    def __init__(self, a, b, node_name):
        """
        Parameters:
        a: node for which a.out is a numpy array
        b: node for which b.out is a numpy array of the same shape as a
        node_name: node's name (a string)
        """
        ## TODO
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.a = a
        self.b = b

    def forward(self):
        self.out = self.a.out + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_a = self.d_out
        d_b = self.d_out
        self.a.d_out += d_a
        self.b.d_out += d_b
        return self.d_out

    def get_predecessors(self):
        return [self.a, self.b]
```

### 3.3

The completed code for `init` follows:

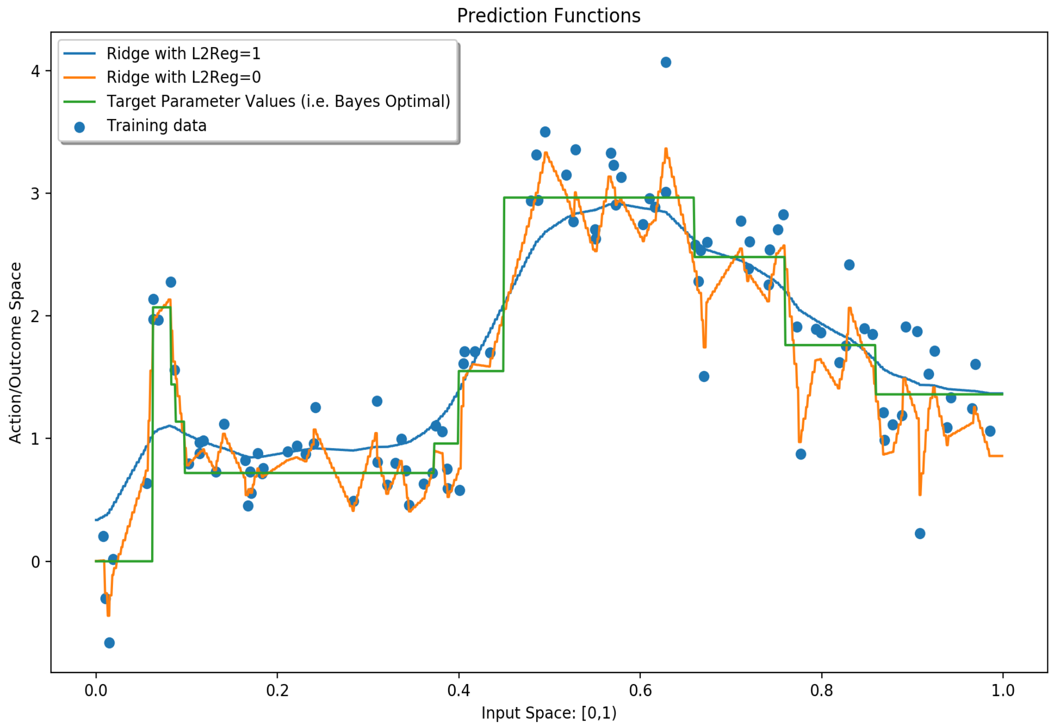
```
def __init__(self, l2_reg=1, step_size=.005,  max_num_epochs = 5000):
        self.max_num_epochs = max_num_epochs
        self.step_size = step_size

        # Build computation graph
        self.x = nodes.ValueNode(node_name="x") # to hold a vector input
        self.y = nodes.ValueNode(node_name="y") # to hold a scalar response
        self.w = nodes.ValueNode(node_name="w") # to hold the parameter vector
        self.b = nodes.ValueNode(node_name="b") # to hold the bias parameter (scalar)
        self.prediction = nodes.VectorScalarAffineNode(x=self.x, w=self.w, b=self.b,
                                                 node_name="prediction")
        #--------------------TODO-------------------
        self.l2reg = l2_reg
        self.square_loss = nodes.SquaredL2DistanceNode(a=self.prediction, b=self.y,
                                               node_name="square loss")
        self.penalty = nodes.L2NormPenaltyNode(l2_reg = self.l2reg, w=self.w,
                                            node_name ="regularization")
        self.objective = nodes.SumNode(a=self.square_loss, b=self.penalty,
                                    node_name="l2 penalized square loss")

        # Group nodes into types to construct computation graph function
        self.inputs = [self.x]
        self.outcomes = [self.y]
        self.parameters = [self.w, self.b]

        self.graph = graph.ComputationGraphFunction(self.inputs, self.outcomes,self.
    parameters, self.prediction,self.objective)
```

With l2Reg $= 1$, the average training loss for the final epoch is 0.1993.
With l2Reg $= 0$, the average training loss for the final epoch is 0.0276.



# 4

## 4.1

### 4.1.1

1. We have,

$$\frac{\partial J}{\partial W_{ij}} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial W_{ij}}.$$

And note that by simplifying the matrix notation we can see that.

$$y_i = W_{i.}x + b_i$$

Then,it follows $\forall r \neq i$,

$$\frac{\partial y_r}{\partial W_{ij}} = 0$$

Then, the sum simplifies to:

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial W_{ij}}.$$

Now, note:

$$y_i = W_{i1}x_1 + W_{i2}x_2 + \cdots + W_{ij}x_j + \cdots + W_{id}x_d + b_i$$

Since we have only one term with $W_{ij}$,

$$\frac{\partial y_i}{\partial W_{ij}} = x_j$$

So, putting it all together,

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i}x_j$$

2. It easily follows from the previous equation that the rate of change for W in matrix form can be written as follows:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} x^T$$

,which is an outer product and results in a matrix.

3. we can start by looking at any coordinate $x_i$:

$$\frac{\partial J}{\partial x_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x_i}$$

Let us find $\frac{\partial y}{\partial x_i}$,
Looking at it component wise for y:

$$y_i = W_{i1}x_1 + W_{i2}x_2 + \cdots + W_{ij}x_j + \cdots + W_{id}x_d + b_i$$

Then,

$$\frac{\partial y_k}{\partial x_i} = W_{ki}$$

Thus,

$$\frac{\partial y}{\partial x_i} = W_{.i}$$

Now, combining for all the coordinate of x, we get:

$$\frac{\partial J}{\partial x} = W^T\left(\frac{\partial J}{\partial y}\right)$$

4. The equation to be proved can be easily seen by applying chain rule:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b}$$
$$\Rightarrow \frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial (Wx+b)}{\partial b}$$

Note that $\frac{\partial (Wx+b)}{\partial b} = 1$ So,

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$$

### 4.1.2

Assuming all the given in the question, and that $J = F(\sigma(A))$
Then, let's start by looking at the rate of change for each coordinate of $A_i$,

$$\frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial S}\frac{\partial S}{\partial A_i}$$

Since $S = \sigma(A)$,

$$\frac{\partial \sigma(A)}{\partial A_i} = (0, 0, \ldots, \sigma'(A - i), \ldots, 0)^T$$

$$\frac{\partial J}{\partial A_i} = (\frac{\partial J}{\partial S})_i . \sigma'(A_i)$$

So, for all coordinates together, it's the Hadamard product:

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$$

### 4.2

1. The completed code for `AffineNode` follows:

```
class AffineNode(object):
    """Node implementing affine transformation (W,x,b)-->Wx+b, where W is a matrix,
    and x and b are vectors
        Parameters:
        W: node for which W.out is a numpy array of shape (m,d)
        x: node for which x.out is a numpy array of shape (d)
        b: node for which b.out is a numpy array of shape (m) (i.e. vector of length
     m)
    """
    ## TODO
    def __init__(self, W, x, b, node_name):

        ## TODO
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.W = W
        self.x = x
        self.b = b

    def forward(self):
        self.out = np.dot(self.W.out, self.x.out) + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        I = self.d_out
        d_w = np.outer(I, self.x.out)
        W_t = np.transpose(self.W.out)
        d_x = np.dot(W_t, I)
        d_b = I
        self.x.d_out += d_x
        self.W.d_out += d_w
        self.b.d_out += d_b
        return self.d_out

    def get_predecessors(self):
        return [self.W, self.x, self.b]
```

2. The completed code for `TanhNode` follows:

```
class TanhNode(object):
    """Node tanh(a), where tanh is applied elementwise to the array a
        Parameters:
        a: node for which a.out is a numpy array
    """
    ## TODO
    def __init__(self, a, node_name):

        ## TODO
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.a = a

    def forward(self):
        self.out = np.tanh(self.a.out)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_a = self.d_out*(1 - np.square(self.out))
        self.a.d_out += d_a
        return self.d_out

    def get_predecessors(self):
        return [self.a]
```

3. The init code for MLP_Regression follows:

```
class MLPRegression(BaseEstimator, RegressorMixin):
    """ MLP regression with computation graph """
    def __init__(self, num_hidden_units=10, step_size=.005, init_param_scale=0.01,
    max_num_epochs = 5000):
        self.num_hidden_units = num_hidden_units
        self.init_param_scale = init_param_scale
        self.max_num_epochs = max_num_epochs
        self.step_size = step_size

        # Build computation graph
        self.x = nodes.ValueNode(node_name="x") # to hold a vector input
        self.y = nodes.ValueNode(node_name="y") # to hold a scalar response
        ## TODO
        self.W1= nodes.ValueNode(node_name="W1") #this is a matrix
        self.b1 = nodes.ValueNode(node_name="b1")
        self.hid1 = nodes.AffineNode(W = self.W1, x=self.x, b=self.b1,
                                node_name="hidden layer 1" )
        self.hid2 = nodes.TanhNode(a = self.hid1, node_name="hidden layer 2")
        self.W2 = nodes.ValueNode(node_name="w2")
        self.b2 = nodes.ValueNode(node_name="b2")
        self.prediction = nodes.VectorScalarAffineNode(x=self.hid2, w=self.W2, b=
    self.b2,
                                                  node_name="prediction")
        self.objective = nodes.SquaredL2DistanceNode(a=self.prediction, b=self.y,
                                    node_name="MLP objective")

        # Group nodes into types to construct computation graph function
        self.inputs = [self.x]
        self.outcomes = [self.y]
        self.parameters = [self.W1, self.b1, self.W2, self.b2]

        self.graph = graph.ComputationGraphFunction(self.inputs, self.outcomes, self
    .parameters, self.prediction, self.objective)
```

The average training error in the last epoch without featurization is 0.231.
The average training error in the last epoch with featurization is 0.048.