Homework 4: Decision Trees, Boosting, and Neural Networks

Decision Tree Implementation

Ans. 1

```python
def compute_entropy(label_array):
    '''
    Calulate the entropy of given label list

    :param label_array: a numpy array of binary labels shape = (n, 1)
    :return entropy: entropy value
    '''
    # count the different labels and their counts in the label array
    label_val, label_count = np.unique(label_array, return_counts= True)
    tot_count = np.sum(label_count)
    label_prob = label_count / tot_count
    epsilon = 1e-12
    entropy = -(label_prob * np.log(label_prob + epsilon)).sum()
    return entropy

def compute_gini(label_array):
    '''
    Calulate the gini index of label list

    :param label_array: a numpy array of labels shape = (n, 1)
    :return gini: gini index value
    '''
    label_val, label_count = np.unique(label_array, return_counts= True)
    tot_count = np.sum(label_count)
    label_prob = label_count / tot_count
    gini = (np.sum(label_prob * (1 - label_prob)))
    return gini
```

please turn over

Ans 2

```
class Decision_Tree(BaseEstimator):

    def __init__(self, split_loss_function, leaf_value_estimator,
                 depth=0, min_sample=5, max_depth=10):
        '''
        Initialize the decision tree classifier

        :param split_loss_function: method with args (X, y) returning loss
        :param leaf_value_estimator: method for estimating leaf value from
array of ys
        :param depth: depth indicator, default value is 0, representing
root node
        :param min_sample: an internal node can be splitted only if it
contains points more than min_smaple
        :param max_depth: restriction of tree depth.
        '''
        self.split_loss_function = split_loss_function
        self.leaf_value_estimator = leaf_value_estimator
        self.depth = depth
        self.min_sample = min_sample
        self.max_depth = max_depth
        self.is_leaf = False
        '''
        ----EDIT: adding additional parameters to the decision tree class--
--
        '''
        self.left = None
        self.right = None
        self.split_id = -1
        self.split_value = None
        self.value = None

    def fit(self, x, y):
        '''
        This should fit the tree classifier by setting the values
self.is_leaf,
        self.split_id (the index of the feature we want ot split on, if
we're splitting),
        self.split_value (the corresponding value of that feature where the
split is),
        and self.value, which is the prediction value if the tree is a leaf
node.  If we are
        splitting the node, we should also init self.left and self.right to
be Decision_Tree
        objects corresponding to the left and right subtrees. These
subtrees should be fit on
        the data that fall to the left and right,respectively, of
self.split_value.
        This is a recurisive tree building procedure.
```

```
        :param X: a numpy array of training data, shape = (n, m)
        :param y: a numpy array of labels, shape = (n, 1)

        :return self
        '''
        # first check if the current node is to be a leaf or has to be
split further
        if((self.depth >= self.max_depth) or (x.shape[0] <
self.min_sample)):
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        # in case the node needs to be split - we need to create the
subsets of data that need to be given to each child node
        self.find_best_feature_split(x, y)

        if self.split_id == -1:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        left_idx = np.where(x[:, self.split_id] <= self.split_value)
        right_idx = np.where(x[:, self.split_id] > self.split_value)

        if np.sum(left_idx) == 0 or np.sum(right_idx) == 0:
            self.split_id = -1
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        # create the left and right subtrees
        self.left = Decision_Tree(self.split_loss_function,
self.leaf_value_estimator, self.depth + 1, self.min_sample, self.max_depth)
        self.right = Decision_Tree(self.split_loss_function,
self.leaf_value_estimator, self.depth + 1, self.min_sample, self.max_depth)

        self.left.fit(x[left_idx], y[left_idx])
        self.right.fit(x[right_idx], y[right_idx])

        return self

    def find_best_split(self, x_node, y_node, feature_id):
        '''
        For feature number feature_id, returns the optimal splitting point
        for data X_node, y_node, and corresponding loss.

        :param x_node: a numpy array of training data, shape = (n_node, m)
        :param y_node: a numpy array of labels, shape = (n_node, 1)
        '''
        min_loss = np.inf
        split_value = np.inf

        x_node_ftr = x_node[:, feature_id]  # Extract the feature column
```

```
        num_samples = len(y_node)

        # Sort the input feature and maintain corresponding labels
        sort_idx = np.argsort(x_node_ftr)
        sorted_x_node_ftr = x_node_ftr[sort_idx]
        sorted_y_node = y_node[sort_idx]

        # Identify unique values in the sorted feature column
        unique_values = np.unique(sorted_x_node_ftr)

        for value in unique_values:
            # Split the data based on the unique value
            l_idx_mask = sorted_x_node_ftr <= value
            r_idx_mask = ~l_idx_mask  # Complement of the left mask

            y_left = sorted_y_node[l_idx_mask]
            y_right = sorted_y_node[r_idx_mask]

            # Skip if either side is empty
            if len(y_left) == 0 or len(y_right) == 0:
                continue

            # Compute loss
            net_loss = (
                (len(y_left) / num_samples) *
self.split_loss_function(y_left) +
                (len(y_right) / num_samples) *
self.split_loss_function(y_right)
            )

            # Update if current loss is better
            if net_loss < min_loss:
                min_loss = net_loss
                split_value = value

        return split_value, min_loss

    def find_best_feature_split(self, x_node, y_node):
        '''
        Returns the optimal feature to split and best splitting point
        for data X_node, y_node.

        ----EDIT: I have edited the shape of param X to (n_node, m) (prvsly
mentioned as (n_node))----
        :param X: a numpy array of training data, shape = (n_node, m)

        :param y: a numpy array of labels, shape = (n_node, 1)
        '''
        min_loss = np.inf
        for i in range(x_node.shape[1]):
            # print(f"find best feature split - feature: {i}")
            curr_feature = i
            curr_feature_min_split_val, curr_feature_min_loss =
self.find_best_split(x_node, y_node, feature_id= curr_feature)
```

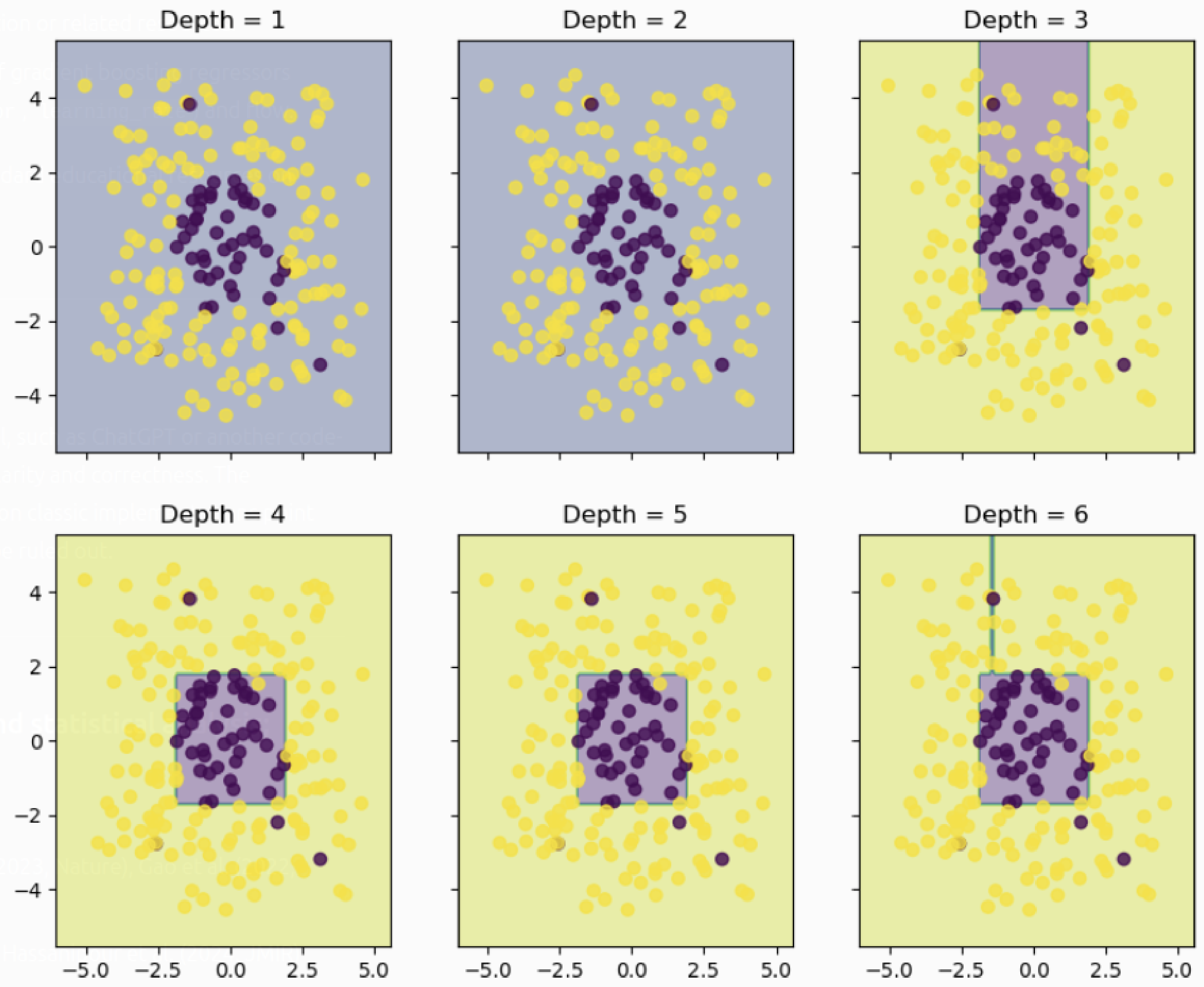
```
            if(min_loss > curr_feature_min_loss):
                self.split_id = curr_feature # curr best feature found
                self.split_value = curr_feature_min_split_val # current
value of best feature on which the split loss is minimum
                min_loss = curr_feature_min_loss

    def predict_instance(self, instance):
        '''
        Predict label by decision tree

        :param instance: a numpy array with new data, shape (1, m)

        :return whatever is returned by leaf_value_estimator for leaf
containing instance
        '''
        if self.is_leaf:
            return self.value
        if instance[self.split_id] <= self.split_value:
            return self.left.predict_instance(instance)
        else:
            return self.right.predict_instance(instance)
```

please turn over

Ans. 3

```
def most_common_label(y):
    '''
    Find most common label
    '''
    label_cnt = Counter(y.reshape(len(y)))
    label = label_cnt.most_common(1)[0][0]
    return label

class Classification_Tree(BaseEstimator, ClassifierMixin):

    loss_function_dict = {
        'entropy': compute_entropy,
        'gini': compute_gini
    }


    def __init__(self, loss_function='entropy', min_sample=5,
max_depth=10):
        '''
        :param loss_function(str): loss function for splitting internal
node
        '''

        self.tree = Decision_Tree(self.loss_function_dict[loss_function],
                                  most_common_label,
                                  0, min_sample, max_depth)

    def fit(self, X, y=None):
        self.tree.fit(X,y)
        return self

    def predict_instance(self, instance):
        value = self.tree.predict_instance(instance)
        return value

# Training classifiers with different depth
clf1 = Classification_Tree(max_depth=1, min_sample=2)
clf1.fit(x_train, y_train_label)

clf2 = Classification_Tree(max_depth=2, min_sample=2)
clf2.fit(x_train, y_train_label)

clf3 = Classification_Tree(max_depth=3, min_sample=2)
clf3.fit(x_train, y_train_label)

clf4 = Classification_Tree(max_depth=4, min_sample=2)
clf4.fit(x_train, y_train_label)

clf5 = Classification_Tree(max_depth=5, min_sample=2)
clf5.fit(x_train, y_train_label)
```

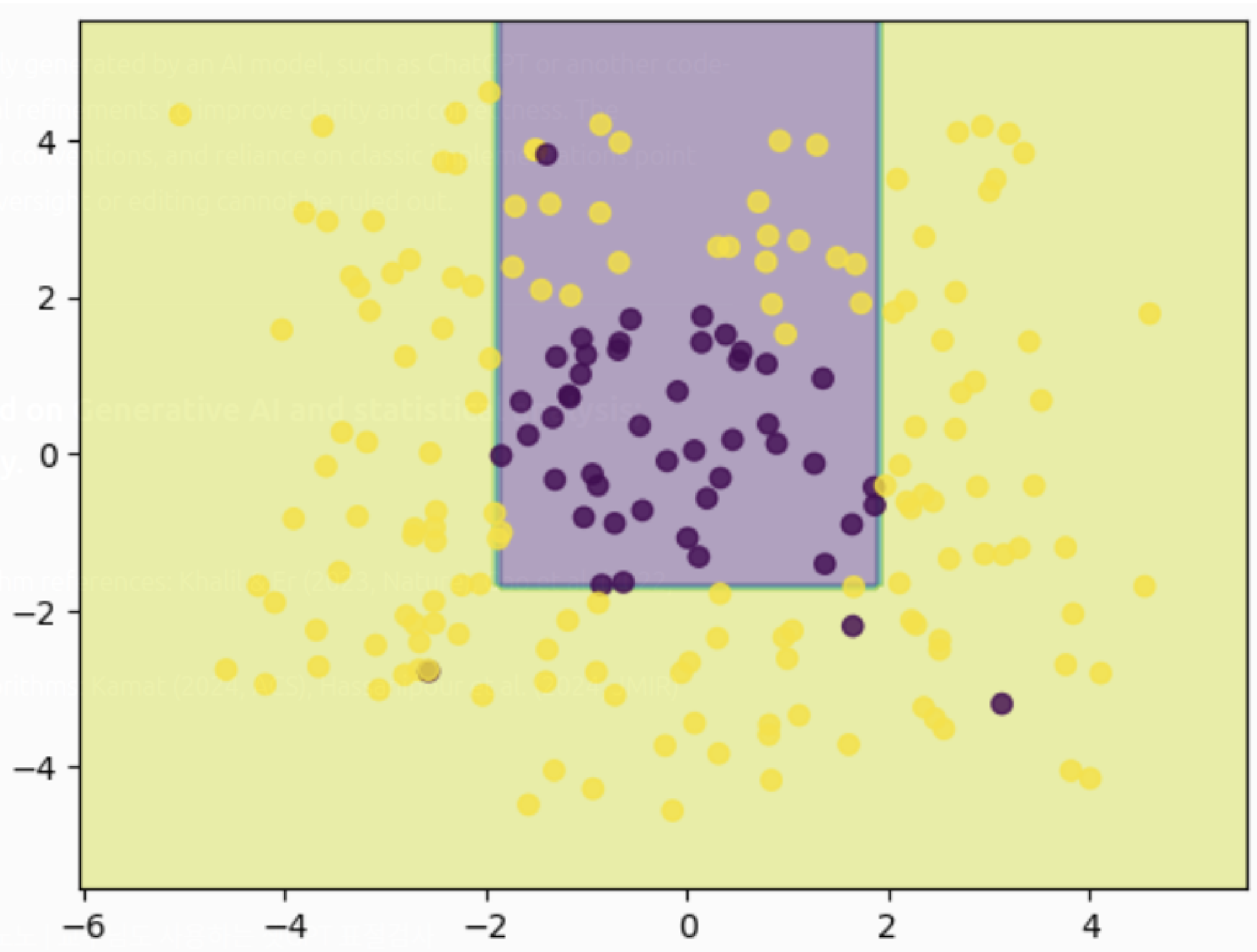
```
clf6 = Classification_Tree(max_depth=6, min_sample=2)
clf6.fit(x_train, y_train_label)

# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', figsize=(10, 8))

for idx, clf, tt in zip(product([0, 1], [0, 1, 2]),
                        [clf1, clf2, clf3, clf4, clf5, clf6],
                        ['Depth = {}'.format(n) for n in range(1, 7)]):

    Z = np.array([clf.predict_instance(x) for x in \
        np.c_[xx.ravel(), yy.ravel()]])
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.4)
    axarr[idx[0], idx[1]].scatter(x_train[:, 0], x_train[:, 1],\
         c=y_train_label[:,0], alpha=0.8)
    axarr[idx[0], idx[1]].set_title(tt)

plt.show()
```



please turn over

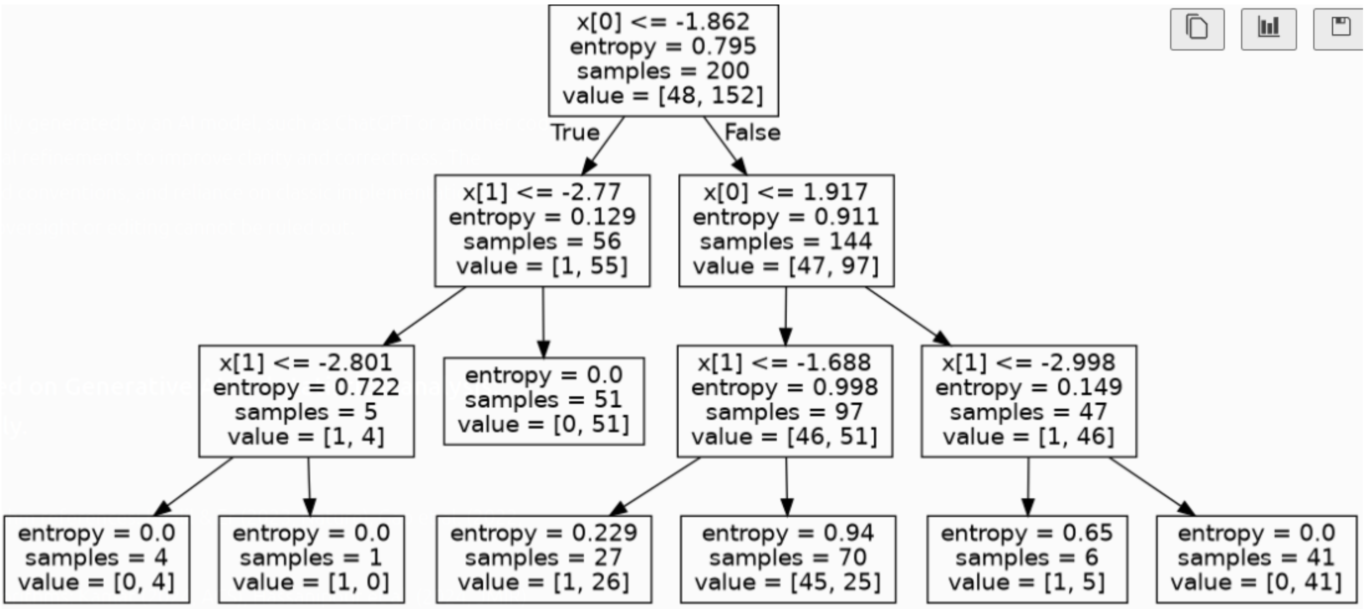
Comparing decision tree with tree model sklearn:

```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=3, \
    min_samples_split=5)
clf.fit(x_train, y_train_label)
export_graphviz(clf, out_file='tree_classifier.dot')


# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = np.array([clf.predict(x[np.newaxis,:]) \
    for x in np.c_[xx.ravel(), yy.ravel()]])
Z = Z.reshape(xx.shape)
plt.figure()
plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(x_train[:, 0], x_train[:, 1],
c=y_train_label[:,0], alpha=0.8)
```



please turn over

```
!dot -Tpng tree_classifier.dot -o tree_classifier.png
Image(filename='tree_classifier.png')
```



please turn over

Ans. 4

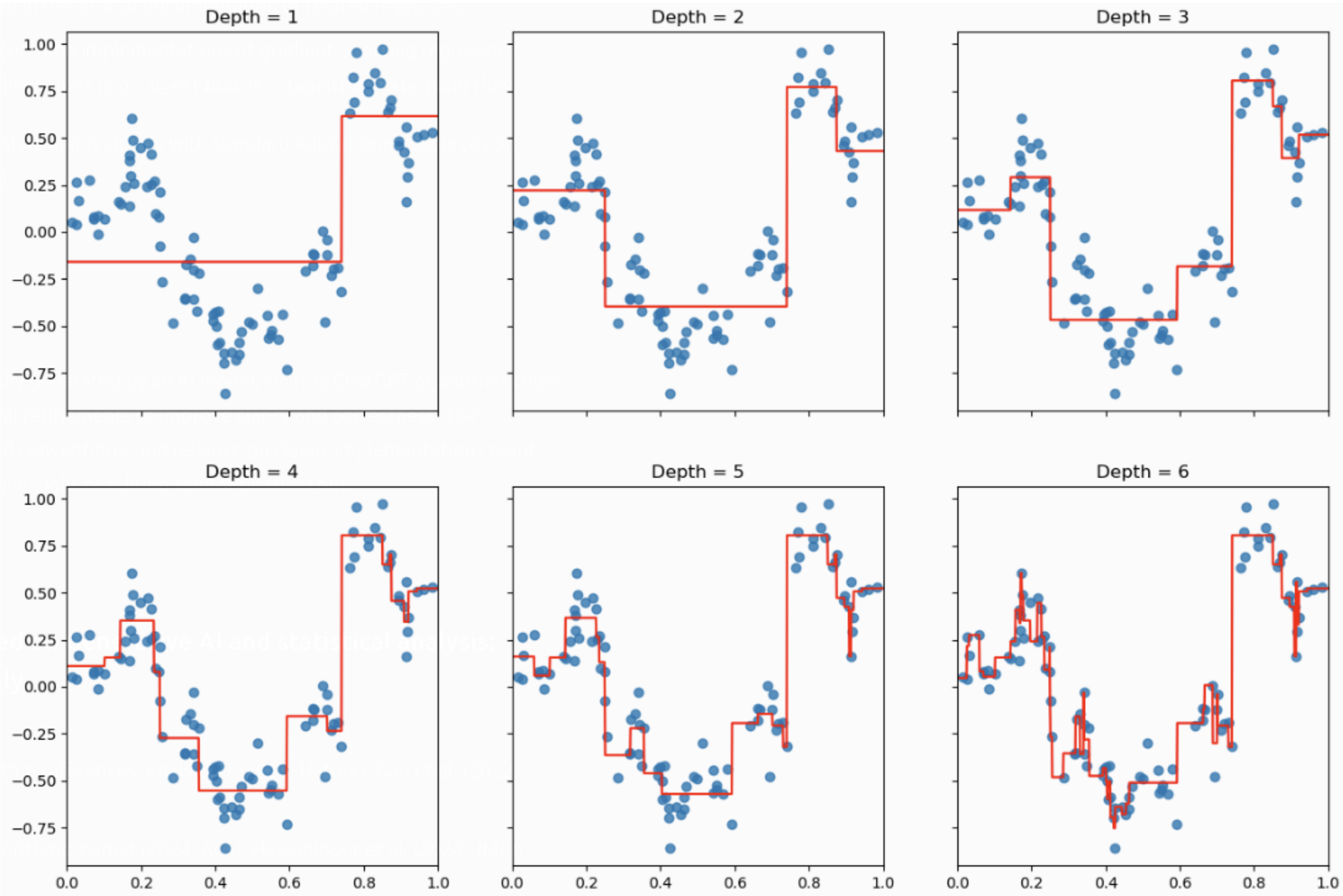
```
# Regression Tree Specific Code
def mean_absolute_deviation_around_median(y):
    '''
    Calulate the mean absolute deviation around the median of a \
        given target list

    :param y: a numpy array of targets shape = (n, 1)
    :return mae
    '''
    median = np.median(y)
    mae = np.mean(np.abs(y - median))

    return mae
```

Regression_Tree class and visualization code

```
class Regression_Tree():
    '''
    :attribute loss_function_dict: dictionary containing the loss \
        functions used for splitting
    :attribute estimator_dict: dictionary containing the estimation \
        functions used in leaf nodes
    '''

    loss_function_dict = {
        'mse': np.var,
        'mae': mean_absolute_deviation_around_median
    }

    estimator_dict = {
        'mean': np.mean,
        'median': np.median
    }

    def __init__(self, loss_function='mse', estimator='mean', \
        min_sample=5, max_depth=10):
        '''
        Initialize Regression_Tree
        :param loss_function(str): loss function used for splitting \
            internal nodes
        :param estimator(str): value estimator of internal node
        '''

        self.tree = Decision_Tree(\
            self.loss_function_dict[loss_function],
                                  self.estimator_dict[estimator],
                                  0, min_sample, max_depth)
```

```
    def fit(self, X, y=None):
        self.tree.fit(X,y)
        return self

    def predict_instance(self, instance):
        value = self.tree.predict_instance(instance)
        return value

data_krr_train = np.loadtxt('krr-train.txt')
data_krr_test = np.loadtxt('krr-test.txt')
x_krr_train, y_krr_train = data_krr_train[:,0].reshape(-1,1),\
    data_krr_train[:,1].reshape(-1,1)
x_krr_test, y_krr_test = data_krr_test[:,0].reshape(-1,1),\
    data_krr_test[:,1].reshape(-1,1)

# Training regression trees with different depth
clf1 = Regression_Tree(max_depth=1,  min_sample=3, \
    loss_function='mae', estimator='mean')
clf1.fit(x_krr_train, y_krr_train)

clf2 = Regression_Tree(max_depth=2,  min_sample=3, \
    loss_function='mae', estimator='mean')
clf2.fit(x_krr_train, y_krr_train)

clf3 = Regression_Tree(max_depth=3,  min_sample=3, \
    loss_function='mae', estimator='mean')
clf3.fit(x_krr_train, y_krr_train)

clf4 = Regression_Tree(max_depth=4,  min_sample=3, \
    loss_function='mae', estimator='mean')
clf4.fit(x_krr_train, y_krr_train)

clf5 = Regression_Tree(max_depth=5,  min_sample=3, \
    loss_function='mae', estimator='mean')
clf5.fit(x_krr_train, y_krr_train)

clf6 = Regression_Tree(max_depth=10,  min_sample=3, \
    loss_function='mae', estimator='mean')
clf6.fit(x_krr_train, y_krr_train)

plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)

f2, axarr2 = plt.subplots(2, 3, sharex='col', \
    sharey='row', figsize=(15, 10))

for idx, clf, tt in zip(product([0, 1], [0, 1, 2]),
                        [clf1, clf2, clf3, clf4, clf5, clf6],
                        ['Depth = {}'.format(n) \
                            for n in range(1, 7)]):

    y_range_predict = np.array([clf.predict_instance(x) \
        for x in x_range]).reshape(-1, 1)
```

```
    axarr2[idx[0], idx[1]].plot(x_range, y_range_predict, color='r')
    axarr2[idx[0], idx[1]].scatter(x_krr_train, y_krr_train, \
        alpha=0.8)
    axarr2[idx[0], idx[1]].set_title(tt)
    axarr2[idx[0], idx[1]].set_xlim(0, 1)
plt.show()
```



please turn over

Ensembling

Ans 5.

```
#Pseudo-residual function.

def pseudo_residual_L2(train_target, train_predict):
    '''
    Compute the pseudo-residual based on current predicted value.
    '''
    return train_target - train_predict

class gradient_boosting():
    '''
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func,
learning_rate=0.01,
                 min_sample=5, max_depth=5):
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators \
            (i.e. number of rounds of gradient boosting)
        :pseudo_residual_func: function used for computing pseudo-residual
between \
            training labels and predicted labels at each iteration
        :param learning_rate: step size of gradient descent
        '''
        self.n_estimator = n_estimator
        self.pseudo_residual_func = pseudo_residual_func
        self.learning_rate = learning_rate
        self.min_sample = min_sample
        self.max_depth = max_depth

        self.estimators = [] #will collect the n_estimator models

    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        :train_data array of inputs of size (n_samples, m_features)
        :train_target array of outputs of size (n_samples,)
        '''
        self.base_model = DecisionTreeRegressor(criterion= 'squared_error',
\
            max_depth= self.max_depth, min_samples_leaf= self.min_sample)
        self.base_model.fit(train_data, train_target)

        for i in range(self.n_estimator):
            pred = self.learning_rate * self.base_model.predict(train_data)
```

```
            for model in self.estimators:
                pred = pred + self.learning_rate * 
model.predict(train_data)

            residual = self.pseudo_residual_func(train_target, pred)

            h_m = DecisionTreeRegressor(max_depth= self.max_depth, \
                min_samples_leaf= self.min_sample)
            h_m.fit(train_data, residual)
            self.estimators.append(h_m)


    def predict(self, test_data):
        '''
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''

        pred = self.base_model.predict(test_data)
        for model in self.estimators:
            pred = pred + (self.learning_rate * model.predict(test_data))
        return pred
```

Ans. 6

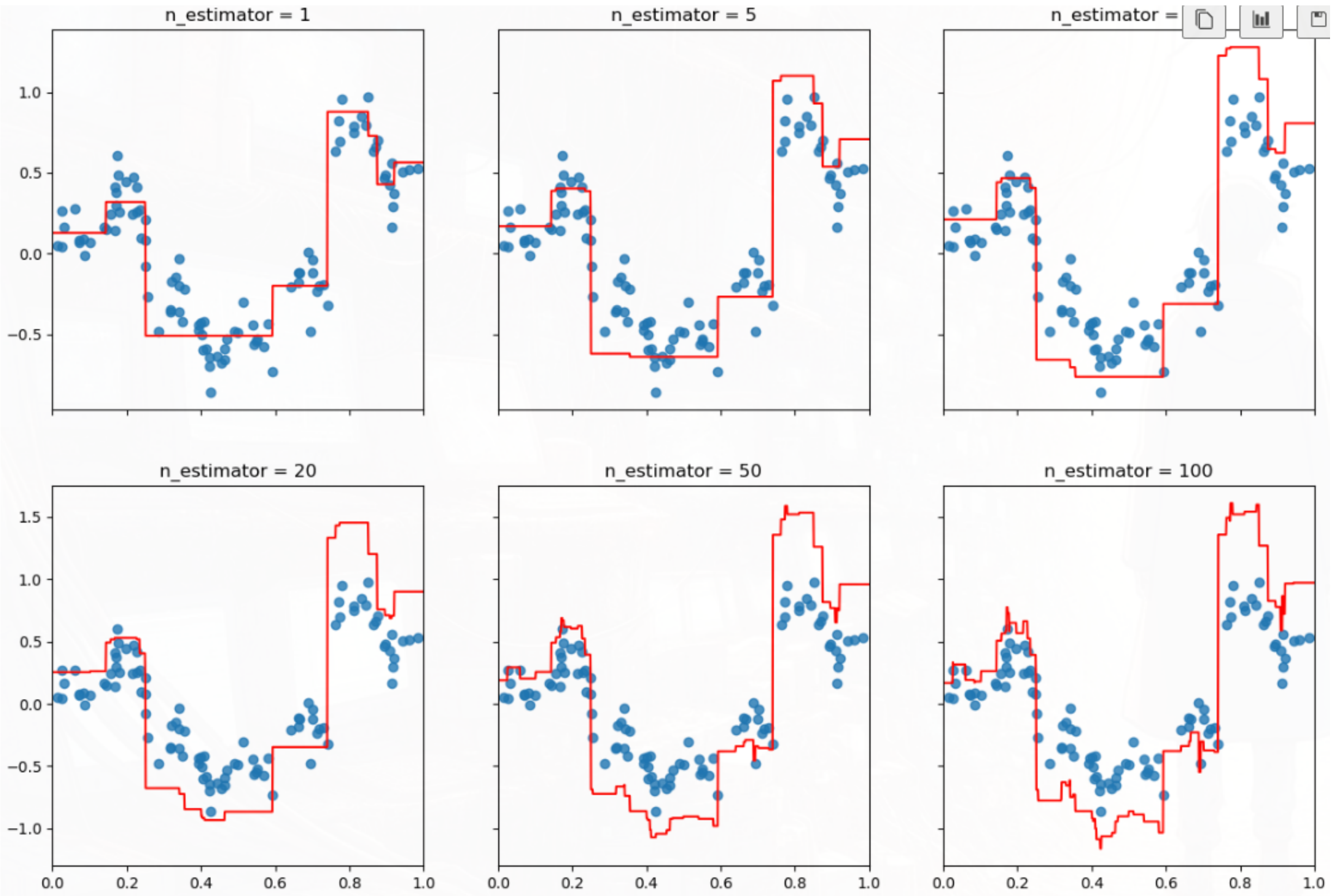
- (NOTE: I have built the gradient_boosting_class using sklearn DecisionTreeRegressor (code provided in the previous question), and using the Regression_Tree class (built upong the self-defined Decision_Tree class) (used in this question). This has been done, because it was unclear to me in Q.s 5 and 6 on what function to use for the gradient boosting models)

```
class gradient_boosting_v2():
    '''
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func, 
learning_rate=0.01,
                 min_sample=5, max_depth=5):
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounds of 
gradient boosting)
        :pseudo_residual_func: function used for computing pseudo-residual 
between training labels and predicted labels at each iteration
        :param learning_rate: step size of gradient descent
        '''
        self.n_estimator = n_estimator
        self.pseudo_residual_func = pseudo_residual_func
        self.learning_rate = learning_rate
```

```
        self.min_sample = min_sample
        self.max_depth = max_depth

        self.estimators = [] #will collect the n_estimator models

    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        :train_data array of inputs of size (n_samples, m_features)
        :train_target array of outputs of size (n_samples,)
        '''
        self.base_model = Regression_Tree(max_depth= self.max_depth,
min_sample= self.min_sample)
        self.base_model.fit(train_data, train_target)

        for i in range(self.n_estimator):
            pred = self.learning_rate *
np.array([self.base_model.predict_instance(instance) for instance in
train_data])
            for model in self.estimators:
                pred = pred + self.learning_rate *
np.array([model.predict_instance(instance) for instance in train_data])

            residual = self.pseudo_residual_func(train_target, pred)

            # h_m = DecisionTreeRegressor(max_depth= self.max_depth,
min_samples_leaf= self.min_sample)
            h_m = Regression_Tree(min_sample= self.min_sample, max_depth=
self.max_depth)
            h_m.fit(train_data, residual)
            self.estimators.append(h_m)


    def predict(self, test_data):
        '''
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''

        pred = np.array([self.base_model.predict_instance(instance) for
instance in test_data])
        for model in self.estimators:
            pred = pred + (self.learning_rate *
np.array([model.predict_instance(instance) for instance in test_data]))
        return pred
```

```
plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)

f2, axarr2 = plt.subplots(2, 3, sharex='col', sharey='row', figsize=(15,
10))
```

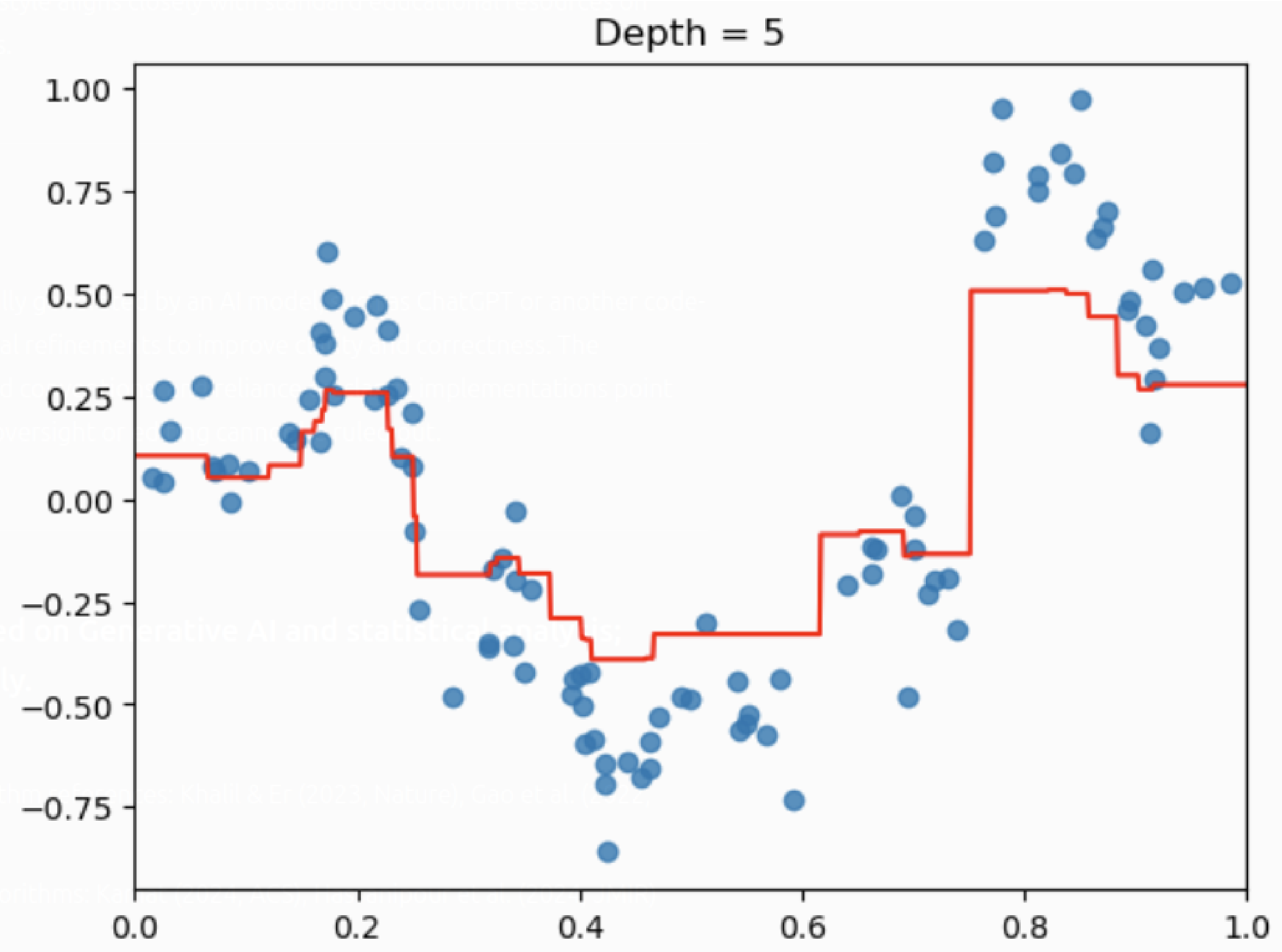
```
for idx, i, tt in zip(product([0, 1], [0, 1, 2]),
                       [1, 5, 10, 20, 50, 100],
                       ['n_estimator = {}'.format(n) \
                        for n in [1, 5, 10, 20, 50, 100]]):

    gbm_1d = gradient_boosting_v2(n_estimator=i,
pseudo_residual_func=pseudo_residual_L2, \
                                  max_depth=3, learning_rate=0.1)
    gbm_1d.fit(x_krr_train, y_krr_train[:,0])

    y_range_predict = gbm_1d.predict(x_range)

    axarr2[idx[0], idx[1]].plot(x_range, y_range_predict, color='r')
    axarr2[idx[0], idx[1]].scatter(x_krr_train, y_krr_train, alpha=0.8)
    axarr2[idx[0], idx[1]].set_title(tt)
    axarr2[idx[0], idx[1]].set_xlim(0, 1)
```



debugging code via sklearn

```
plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)
max_depth = 5

clf = GradientBoostingRegressor(criterion='squared_error', \
    max_depth=max_depth, min_samples_leaf=5, learning_rate= 0.01,
```

```
    n_estimators= 100)
    clf.fit(x_krr_train, y_krr_train)
    y_range_predict = np.array([clf.predict(x[np.newaxis,:]) for x in
    x_range]).reshape(-1, 1)

    plt.plot(x_range, y_range_predict, color='r')
    plt.scatter(x_krr_train, y_krr_train, alpha=0.8)
    plt.title('Depth = {}'.format(max_depth))
    plt.xlim(0, 1)
    plt.show()
```



please turn over

Neural Network Introduction, Computation Graph Framework

- No questions asked here `please turn over`

Ridge Regression

Ans. 7

```python
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 numpy scalar array (e.g. np.array(.01)) (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

    def forward(self):
        w_vec = self.w.out
        w_norm = np.linalg.norm(w_vec)
        self.out = ((w_norm * w_norm) * self.l2_reg)
        self.d_out = np.zeros_like(self.out)
        return self.out

    def backward(self):
        w_vec = self.w.out
        self.w.d_out = 2 * self.l2_reg * w_vec * self.d_out

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

```
● (deep_learning) (base) arjun-prasad@arjun-prasad-Lenovo-Legion-5-15ARH05:~/ARJUN/CourseWork/Mach
  DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 4.190667061846293e-09.
```

please turn over

Ans 8.

```
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)
        """
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.b = b
        self.a = a

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

    def backward(self):
        d_a = 1 * self.d_out
        d_b = 1 * self.d_out
        self.a.d_out += d_a
        self.b.d_out += d_b

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

```
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 1.636578803425771e-09.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 5.838672347430118e-10.
```

please turn over

Ans 9.

```
class RidgeRegression(BaseEstimator, RegressorMixin):
    """ Ridge regression with computation graph """
    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")
        self.objective = nodes.SumNode(nodes.SquaredL2DistanceNode(\
            a=self.prediction, b=self.y, \
            node_name="square loss"), \
            nodes.L2NormPenaltyNode(l2_reg, self.w, "l2_reg"), \
            "ridge_reg")

        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)


    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)

        init_parameter_values = {"w": np.zeros(num_ftrs), "b":
np.array(0.0)}
        self.graph.set_parameters(init_parameter_values)

        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
                obj, grads = self.graph.get_gradients(input_values = {"x":
X[j]},
                                                    outcome_values = {"y":
y[j]})
                #print(obj)
                epoch_obj_tot += obj
```

```
                # Take step in negative gradient direction
                steps = {}
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                self.graph.increment_parameters(steps)

            if epoch % 50 == 0:
                train_loss = sum((y - self.predict(X,y)) **2)/num_instances
                print("Epoch ",epoch,": Ave objective=",\
                    epoch_obj_tot/num_instances,\
                        " Ave training loss: ",train_loss)

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting
data!")

        num_instances = X.shape[0]
        preds = np.zeros(num_instances)
        for j in range(num_instances):
            preds[j] = self.graph.get_prediction(input_values={"x":X[j]})

        return preds



def main():
    data_fname = "data.pickle"
    x_train, y_train, x_val, y_val, target_fn, coefs_true, \
        featurize = setup_problem.load_problem(data_fname)

    # Generate features
    X_train = featurize(x_train)
    X_val = featurize(x_val)

    pred_fns = []
    x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))
    X = featurize(x)

    l2reg = 1
    estimator = RidgeRegression(l2_reg=l2reg, \
        step_size=0.00005, max_num_epochs=2000)
    estimator.fit(X_train, y_train)
    name = "Ridge with L2Reg="+str(l2reg)
    pred_fns.append({"name":name, "preds": estimator.predict(X) })


    l2reg = 0
    estimator = RidgeRegression(l2_reg=l2reg, \
        step_size=0.0005, max_num_epochs=500)
    estimator.fit(X_train, y_train)
    name = "Ridge with L2Reg="+str(l2reg)
```

```
    pred_fns.append({"name":name, "preds": estimator.predict(X) })

    # Let's plot prediction functions and compare coefficients for several
fits
    # and the target function.

    pred_fns.append({"name": "Target Parameter Values (i.e. Bayes
Optimal)", \
        "coefs": coefs_true, "preds": target_fn(x)})

    plot_utils.plot_prediction_functions(x, pred_fns, x_train, \
        y_train, legend_loc="best")

if __name__ == '__main__':
  main()
```

- test cases

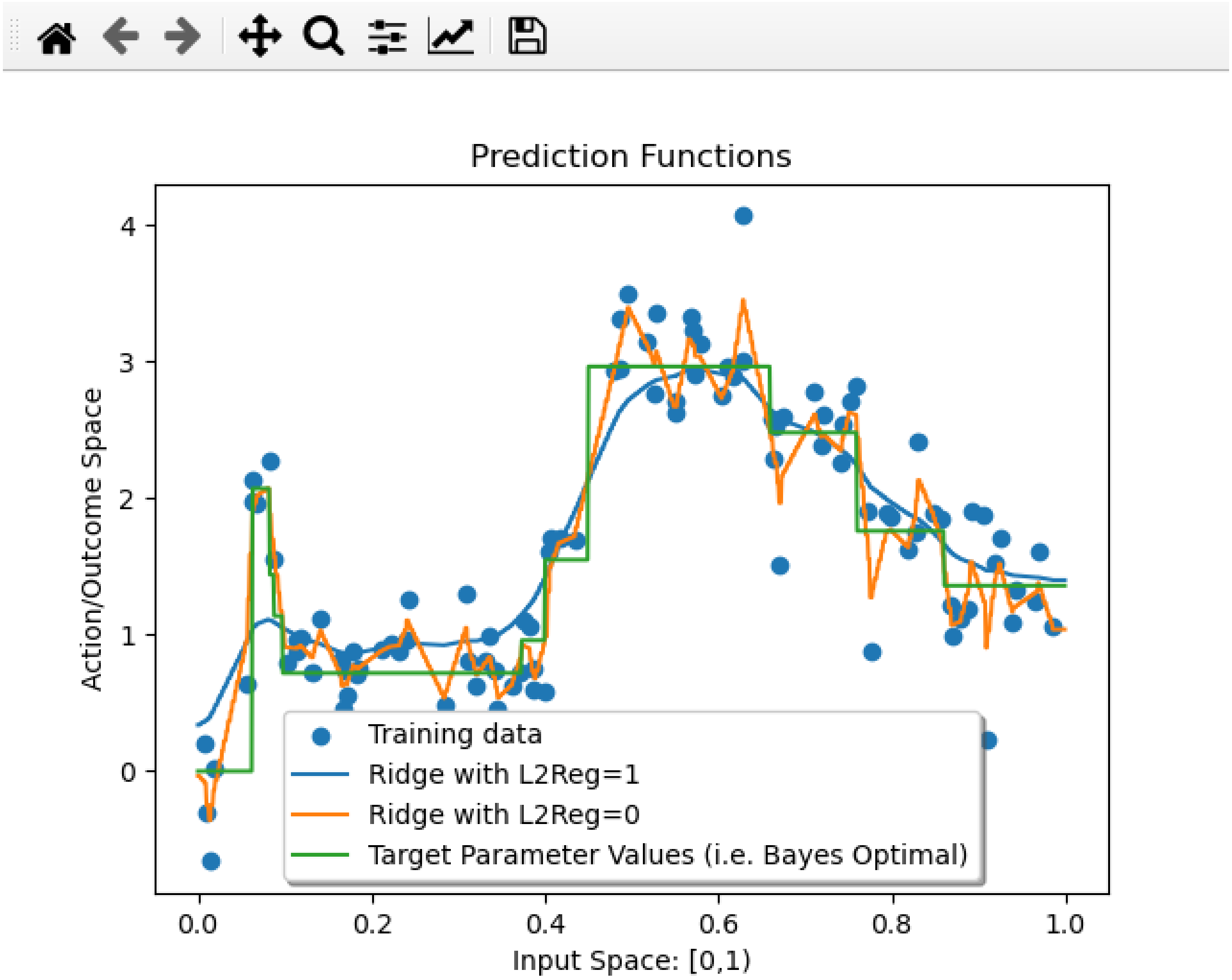
```
(deep_learning) (base) arjun-prasad@arjun-prasad-Lenovo-Legion-5-15ARH05:~/ARJUN/Cour
idge_regression.t.py
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 5.628854579506
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 5.838671770815658
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 1.636578859934555
.DEBUG: (Parameter w) Max rel error for partial deriv 2.7820614265556677e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 7.940832693784278e-10.
```

- Average Training Loss:

```
Epoch  1800 : Ave objective= 0.30422838177646816  Ave training loss:  0.19951172464135844
Epoch  1850 : Ave objective= 0.3051244963548033  Ave training loss:  0.19948279848411762
Epoch  1900 : Ave objective= 0.30458070045385155  Ave training loss:  0.20049564456872507
Epoch  1950 : Ave objective= 0.3049240603230615  Ave training loss:  0.19998102561296593
Epoch  0 : Ave objective= 0.6909674276953551  Ave training loss:  0.5351599062284119
Epoch  50 : Ave objective= 0.11900738345274707  Ave training loss:  0.10623783437136644
Epoch  100 : Ave objective= 0.10561873811273172  Ave training loss:  0.08500256731032349
Epoch  150 : Ave objective= 0.08378111519194606  Ave training loss:  0.07946335016874015
Epoch  200 : Ave objective= 0.0718557501028382  Ave training loss:  0.06497773902021574
Epoch  250 : Ave objective= 0.07118748491708027  Ave training loss:  0.06323625558006388
Epoch  300 : Ave objective= 0.05812454069791321  Ave training loss:  0.05272421376954614
Epoch  350 : Ave objective= 0.05478289328024275  Ave training loss:  0.05396470524099741
Epoch  400 : Ave objective= 0.05430683088238369  Ave training loss:  0.05080130349572671
Epoch  450 : Ave objective= 0.050543836008525085  Ave training loss:  0.05287872115369579
```

  - Avg training loss 1, l2reg = 1 => 0.19998102561296593
  - Avg training loss 2, l2reg = 0 => 0.05287872115369579

- output image generated on running `python ridge_regression.py`



`please turn over`

Multilayer Perceptron

Ans 10.

(10) To prove: $\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i} x_j$ $\quad\Big\|\quad$ $x = (x_1, x_2, \dots, x_d)^T$

Proof:

for 'n' samples

$$\frac{\partial J}{\partial w_{ij}} = \sum_{k=1}^{n} \frac{\partial J}{\partial y_k} \cdot \frac{\partial y_k}{\partial w_{ij}}$$

$$= \underbrace{0}_{k \neq i} + \overbrace{\frac{\partial J}{\partial y_i}\left(\frac{\partial y_i}{\partial w_{ij}}\right)}^{k=i} \qquad (\forall k \in \{1, 2, \dots, n\})$$

$$\frac{\partial y_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{j=1}^{d} w_{ij} x_j + b_i\right)$$

$$= x_j$$

Therefore

$$\boxed{\frac{\partial J}{\partial w_{ij}} = 0 + \frac{\partial J}{\partial y_i} x_j}$$

↳ Hence Proved.

please turn over

Ans 11.

(11) $\frac{\partial J}{\partial y} \in \mathbb{R}^{m \times 1}$, i^{th} entry $\Rightarrow \frac{\partial J}{\partial y_i}$

$\frac{\partial J}{\partial W} \in \mathbb{R}^{m \times d}$, i,j^{th} entry $\Rightarrow \frac{\partial J}{\partial W_{ij}}$

We know that for a given sample x,

$$\left.\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i} \cdot x_j \right\} \text{(from prob. 10)}.$$

$\therefore$ for $X \in \mathbb{R}^{m \times d}$ (m samples, d features)

we can write

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i} \cdot x_{ij}$$

Thus

$$\underbrace{\left(\frac{\partial J}{\partial w}\right)^T}_{d \times m} = \underbrace{X}_{m \times d} \cdot \underbrace{\left(\frac{\partial J}{\partial y}\right)^T}_{m \times 1}$$

$$\Rightarrow \boxed{\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y} \cdot X^T} \qquad \left[(AB)^T = B^T A^T\right]$$

$\llcorner$ (outer product)

please turn over

Ans 12.

(12) $\frac{\partial J}{\partial x} \in \mathbb{R}^d$

$$\frac{\partial J}{\partial x} = \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_d} \end{bmatrix}$$

$$\frac{\partial J}{\partial x_k} = \sum_{i=1}^{m} \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial x_k}.$$

$$y_i = \sum_{l=1}^{d} W_{il} x_l + b_i$$

$$\Rightarrow \frac{\partial y_i}{\partial x_k} = \sum_{l=1}^{d} \frac{\partial (W_{il} x_l)}{\partial x_k} + b_i^0$$

$$\boxed{\frac{\partial y_i}{\partial x_k} = W_{ik}.}$$

$$\therefore \frac{\partial J}{\partial x_k} = \sum_{i=1}^{m} \frac{\partial J}{\partial y_i} \cdot W_{ik}.$$

$$\Rightarrow \boxed{\frac{\partial J}{\partial x} = W^T \frac{\partial J}{\partial y}}$$ Hence proved

please turn over

Ans 13.

13

$$y = wx + b$$

$$\Rightarrow \frac{\partial y}{\partial b} = 1.$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \cdot \cancel{\frac{\partial y}{\partial b}} \rightarrow 1$$

$$\boxed{\partial J / \partial b = \partial J / \partial y}$$ Hence proved.

please turn over

Ans 14.

14 $S = \sigma(A)$

for i^{th} dimension of A, we have A_i.

$\therefore$

$$\frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial(\sigma(A_i))} \cdot \frac{\partial(\sigma(A_i))}{\partial A}$$

$$= \frac{\partial J}{\partial S_i} \cdot \sigma'(A_i)$$

$$\Rightarrow \boxed{\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S_i} \odot \sigma'(A_i)}$$

please turn over

Ans 15.

```
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)
    """
    def __init__(self, w, x, b, node_name):
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.x = x
        self.w = w
        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):
        d_x = np.dot(self.w.out.T,self.d_out)
        d_b = self.d_out
        d_w = np.outer(self.d_out, self.x.out)
        self.x.d_out += d_x
        self.w.d_out += d_w
        self.b.d_out += d_b


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

test results

```
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 5.298097672432879e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 1.694796002711163e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 1.6365789068091537e-09.
```

please turn over

Ans. 16

```
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
    """
    def __init__(self, a, node_name):
        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 = (1-self.out**2)*self.d_out
        self.a.d_out += d_a
        return self.d_out

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

- test results

```
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 3.647110392742941e-08.
```

please turn over

Ans. 17

```
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

        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.w1 = nodes.ValueNode(node_name="W1") # to hold the parameter
vector
        self.b1 = nodes.ValueNode(node_name=f"b1") # to hold the bias
parameter (scalar)

        self.w2 = nodes.ValueNode(node_name="w2") # to hold the parameter
vector
        self.b2 = nodes.ValueNode(node_name=f"b2")


        self.prediction = nodes.AffineNode(x = self.x, w = self.w1, \
            b = self.b1,node_name="hidden_layer")
        self.prediction = nodes.TanhNode(a = self.prediction,
node_name=f"activation")

        self.prediction = nodes.VectorScalarAffineNode(x = self.prediction
, \
            w=self.w2,b = self.b2, node_name="output_layer")

        self.objective = nodes.SquaredL2DistanceNode(a=self.prediction, \
            b=self.y, node_name="square loss")

        # Group nodes into types to construct computation graph function
        self.inputs = [self.x]
        self.outcomes = [self.y]
        self.parameters = [self.w1,self.w2, self.b1,self.b2]

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

    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)
        s = self.init_param_scale
        init_values = {"W1": s *
np.random.standard_normal((self.num_hidden_units, \
            num_ftrs)),
                       "b1": s *
np.random.standard_normal((self.num_hidden_units)),
```

```
                             "w2": s *
np.random.standard_normal((self.num_hidden_units)),
                             "b2": s * np.array(np.random.randn()) }

        self.graph.set_parameters(init_values)

        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
                obj, grads = self.graph.get_gradients(input_values = {"x":
X[j]},
                                                    outcome_values = {"y":
y[j]})
                #print(obj)
                epoch_obj_tot += obj
                # Take step in negative gradient direction
                steps = {}
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                self.graph.increment_parameters(steps)
                #pdb.set_trace()

            if epoch % 50 == 0:
                train_loss = sum((y - self.predict(X,y)) **2)/num_instances
                print("Epoch ",epoch,": Ave objective=",\
                    epoch_obj_tot/num_instances," Ave training loss:
",train_loss)

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting
data!")

        num_instances = X.shape[0]
        preds = np.zeros(num_instances)
        for j in range(num_instances):
            preds[j] = self.graph.get_prediction(input_values={"x":X[j]})

        return preds
```

- test cases

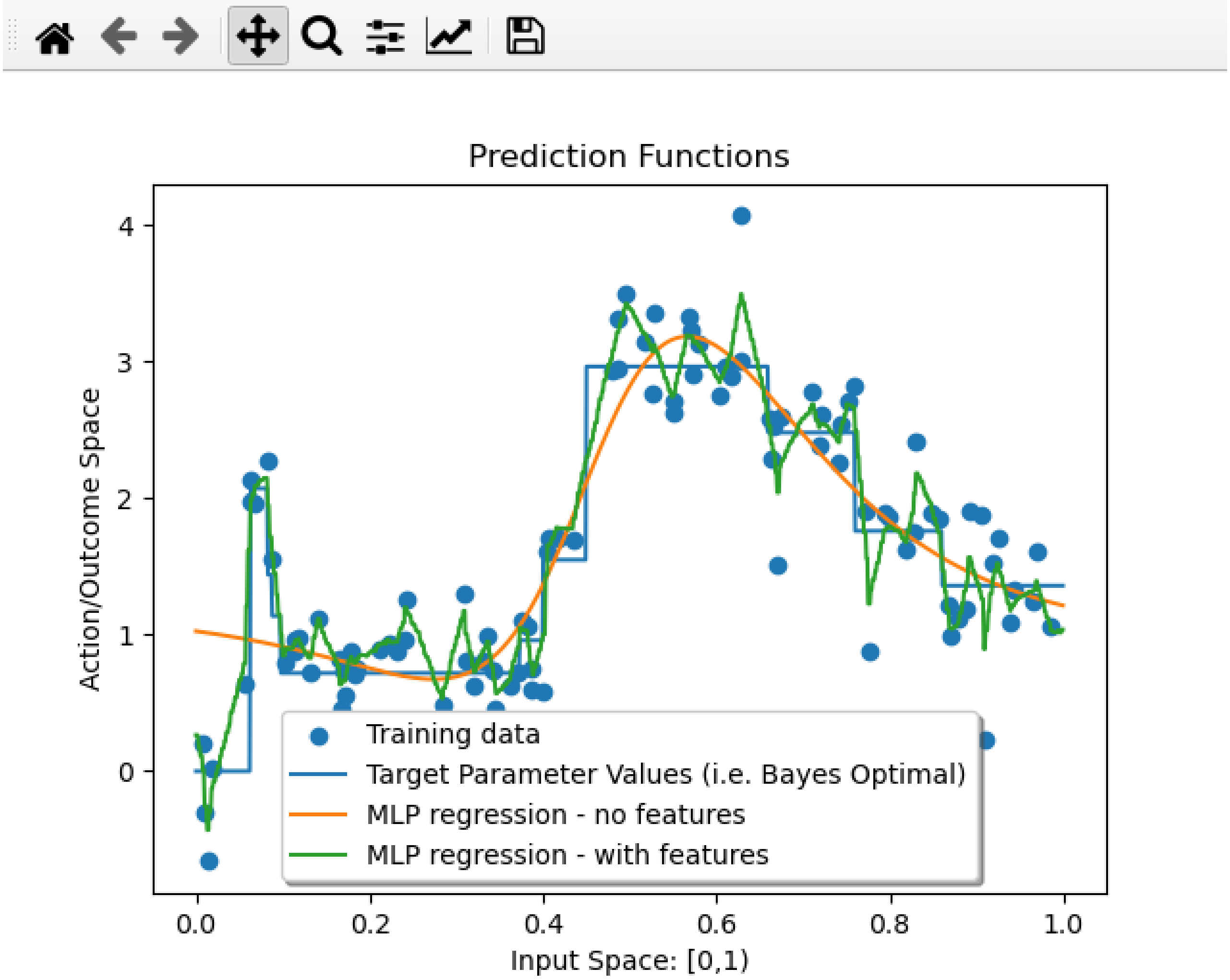
```
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 5.298097672432879e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 1.694796002711163e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 1.6365789068091537e-09.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 3.647110392742941e-08.
.DEBUG: (Parameter W1) Max rel error for partial deriv 1.7553991632627748e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv 2.1649821289952703e-07.
DEBUG: (Parameter w2) Max rel error for partial deriv 1.464398757659339e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 5.042356208756589e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

- running `python mlp_regression.py`

```
Epoch  4800 : Ave objective= 0.24417850889609855  Ave training loss:  0.23907543751001983
Epoch  4850 : Ave objective= 0.24038347369486165  Ave training loss:  0.24084079771282443
Epoch  4900 : Ave objective= 0.24243009939525678  Ave training loss:  0.23829217509160958
Epoch  4950 : Ave objective= 0.24175358009397616  Ave training loss:  0.2375514899589099
Epoch  0 : Ave objective= 3.163273548673171  Ave training loss:  2.5605331414531265
Epoch  50 : Ave objective= 0.14844708437599397  Ave training loss:  0.1397147611961311
Epoch  100 : Ave objective= 0.1116268144045757  Ave training loss:  0.1223455989479196
Epoch  150 : Ave objective= 0.1025335662192338  Ave training loss:  0.09111530930261971
Epoch  200 : Ave objective= 0.08920208778374379  Ave training loss:  0.08274506292670711
Epoch  250 : Ave objective= 0.0770484544738569  Ave training loss:  0.06741359647846575
Epoch  300 : Ave objective= 0.06939536180896462  Ave training loss:  0.06978848314951241
Epoch  350 : Ave objective= 0.05688447211810779  Ave training loss:  0.07793590268056473
Epoch  400 : Ave objective= 0.056197564706299934  Ave training loss:  0.04543237206017478
Epoch  450 : Ave objective= 0.05079795148881582  Ave training loss:  0.04350373562087348
```

  - avg training error 1 = 0.2375514899589099
  - avg training error 2 = 0.04350373562087348

- output generated on running `python mlp_regression.py`



`please turn over`

- *NOTE: optional questions 18, 19, and 20 have not been added to this pdf*