Homework 6 ML Solutions

1 Decision Tree Implementation

Q1. Complete the compute_entropy and compute_gini functions.

```
from scipy.stats import entropy
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
    '''
    label_array = label_array.flatten().tolist()#label_array.reshape(1, -1)
    #label_array = label_array.tolist()
    #label_array = label_array[0]
    labels = list(set(label_array))
    counts = {}
    for label in labels:
        counts[label] = 0
    tot = len(label_array)
    for label in label_array:
        counts[label]+=1
    ent = 0
    for label in labels:
        p_label = float(counts[label])/tot
        ent += -1.0*p_label*np.log2(p_label)

    #scipy_ent = entropy(label_array, base=2)
    #print(ent, scipy_ent)
    return ent
    # Your code goes here

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
    '''
    # Your code goes here
    label_array = label_array.flatten().tolist()
    #label_array = label_array.reshape(1, -1)
    #label_array = list(label_array)
    labels = list(set(label_array))
    counts = {}
    for label in labels:
        counts[label] = 0
    tot = len(label_array)
    for label in label_array:
        counts[label]+=1
    sum_p_2 = 0
    for label in labels:
        p_label = float(counts[label])/tot
        sum_p_2 += p_label**2
    gini = 1 - sum_p_2
    return gini
```

Q2. Complete the class Decision Tree, given in the skeleton code

```
class Decision_Tree(BaseEstimator):

    def __init__(self, split_loss_function, leaf_value_estimator,
                 depth=0, min_sample=5, max_depth=10):
        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

    def fit(self, x, y):
        if self.depth == self.max_depth or len(y) <= self.min_sample:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        split_id, split_value = self.find_best_feature_split(x, y)
        if split_id is None:
            self.is_leaf = True
```

```
            self.value = self.leaf_value_estimator(y)
            return self
        else:
            left_X = []
            left_y = []
            right_X = []
            right_y = []

            assert split_id is not None and split_value is not None

            for i in range(len(x)):
                if x[i][split_id] <= split_value:
                    left_X.append(x[i])
                    left_y.append(y[i])
                else:
                    right_X.append(x[i])
                    right_y.append(y[i])


            left_X = np.asarray(left_X)
            left_y = np.asarray(left_y).reshape(-1,1)
            right_X = np.asarray(right_X)
            right_y = np.asarray(right_y).reshape(-1,1)

            #print(left_y.reshape(1, -1), right_y.reshape(1, -1))

            if len(left_X)>0:
                self.left = Decision_Tree(self.split_loss_function,
                    self.leaf_value_estimator, self.depth+1, self.min_sample,
                        self.max_depth)
                self.left.fit(left_X, left_y)

            if len(right_X)>0:
                self.right = Decision_Tree(self.split_loss_function,
                    self.leaf_value_estimator, self.depth+1, self.min_sample,
                        self.max_depth)
                self.right.fit(right_X, right_y)
            return self

    def find_best_split(self, x_node, y_node, feature_id):
        temp = np.hstack((x_node, y_node))
        temp = np.array(sorted(temp, key=lambda row: row[feature_id]))
        best_loss = None
        best_split_value = None
```

```
        for idx in range(x_node.shape[0]-1):
            y_left = temp[:idx + 1, -1].reshape(-1,1)
            y_right = temp[idx + 1:, -1].reshape(-1,1)
            loss_left = (1.0*len(y_left)/len(y_node)) *
                self.split_loss_function(y_left)
            loss_right = (1.0*len(y_right)/len(y_node)) *
                self.split_loss_function(y_right)
            curr_loss = loss_left+loss_right
            if best_loss is None or curr_loss < best_loss:
                best_loss = curr_loss
                best_split_value = temp[idx][feature_id]

        return best_split_value, best_loss

    def find_best_feature_split(self, x_node, y_node):
        best_loss = self.split_loss_function(y_node)
        #print(best_loss)
        best_split_value = None
        best_split_id = None

        for feature_id in range(x_node.shape[1]):
            split_value, feature_loss = self.find_best_split(x_node, y_node,
                feature_id)
            if feature_loss < best_loss:
                best_loss = feature_loss
                best_split_value = split_value
                best_split_id = feature_id

        self.split_id = best_split_id
        self.split_value = best_split_value
        return best_split_id, best_split_value

    def predict_instance(self, instance):
        if self.is_leaf:
            return self.value#entropy
        if instance[self.split_id] <= self.split_value:
            return self.left.predict_instance(instance)
        else:
            return self.right.predict_instance(instance)
```

Q3. Run the code provided that builds trees for the two-dimensional classification data. Include the results

Fig. 1 is the required plot of the obtained results.

Relevant Code:

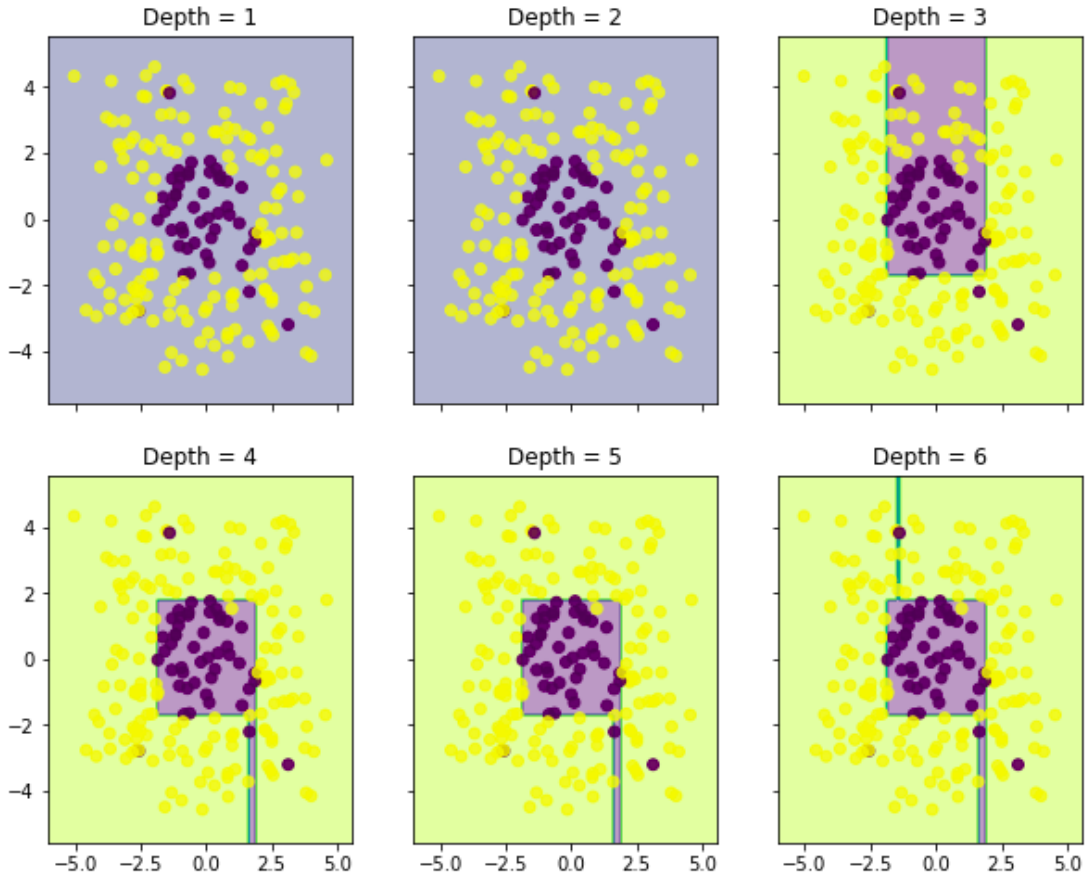

Figure 1: Results for trees on 2D classification

```
def most_common_label(y):
    label_cnt = Counter(y.reshape(len(y)))
    print(label_cnt)
    if label_cnt[0] == label_cnt[1]:
        return 0
    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):
        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)
print("-"*20)
clf2 = Classification_Tree(max_depth=2, min_sample=2)
clf2.fit(x_train, y_train_label)
print("-"*20)

clf3 = Classification_Tree(max_depth=3, min_sample=2)
clf3.fit(x_train, y_train_label)
print("-"*20)

clf4 = Classification_Tree(max_depth=4, min_sample=2)
clf4.fit(x_train, y_train_label)
print("-"*20)

clf5 = Classification_Tree(max_depth=5, min_sample=2)
clf5.fit(x_train, y_train_label)
print("-"*20)
```

```
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()
```

Relevant Output from Notebook:

```
Counter({1: 55, 0: 1})
Counter({1: 97, 0: 47})
--------------------
Counter({1: 4, 0: 1})
Counter({1: 51})
Counter({1: 51, 0: 46})
Counter({1: 46, 0: 1})
--------------------
Counter({1: 4})
Counter({0: 1})
Counter({1: 51})
Counter({1: 26, 0: 1})
Counter({0: 45, 1: 25})
Counter({1: 5, 0: 1})
Counter({1: 41})
--------------------
Counter({1: 4})
```

```
Counter({0: 1})
Counter({1: 51})
Counter({1: 25})
Counter({1: 1, 0: 1})
Counter({0: 44, 1: 2})
Counter({1: 23, 0: 1})
Counter({1: 5})
Counter({0: 1})
Counter({1: 41})
--------------------
Counter({1: 4})
Counter({0: 1})
Counter({1: 51})
Counter({1: 25})
Counter({1: 1, 0: 1})
Counter({1: 1, 0: 1})
Counter({0: 43, 1: 1})
Counter({1: 5, 0: 1})
Counter({1: 18})
Counter({1: 5})
Counter({0: 1})
Counter({1: 41})
--------------------
Counter({1: 4})
Counter({0: 1})
Counter({1: 51})
Counter({1: 25})
Counter({1: 1, 0: 1})
Counter({1: 1, 0: 1})
Counter({0: 41})
Counter({0: 2, 1: 1})
Counter({1: 5})
Counter({0: 1})
Counter({1: 18})
Counter({1: 5})
Counter({0: 1})
Counter({1: 41})
```

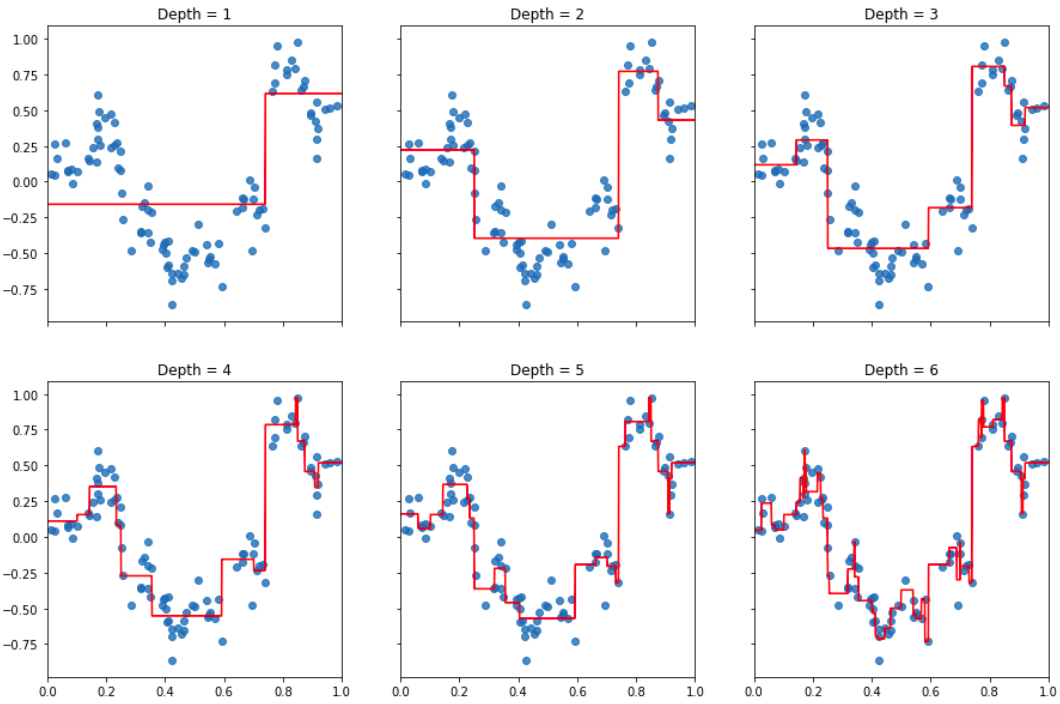

Figure 2: Regression Tree Output

Q4. Complete the function mean absolute deviation around median (MAE). Use the code provided to fit the Regression Tree to the krr dataset using both the MAE loss and median predictions. Include the plots for the 6 fits.

Fig. 2 is the obtained plot. The relevant code follows. **Relevant Code:**

```
def mean_absolute_deviation_around_median(y):
    mae = np.mean(np.abs(y-np.median(y)))
    return mae

class Regression_Tree():
    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):
        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()
```

2 Gradient Boosting Regression Implementation

Q5. Complete the gradient boosting class.

```
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 bo
        :pseudo_residual_func: function used for computing pseudo-residual between tra
        :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,)
        '''
        # Your code goes here
        train_predict = np.zeros(train_target.shape)
        for m in range(self.n_estimator):
            estimator = DecisionTreeRegressor(criterion='mse',
                                              max_depth=self.max_depth,
                                              min_samples_split=self.min_sample)
            pseudo_residuals = self.pseudo_residual_func(train_target, train_predict)

            estimator.fit(train_data, pseudo_residuals)

            train_predict += self.learning_rate * estimator.predict(train_data)
            self.estimators.append(estimator)
```

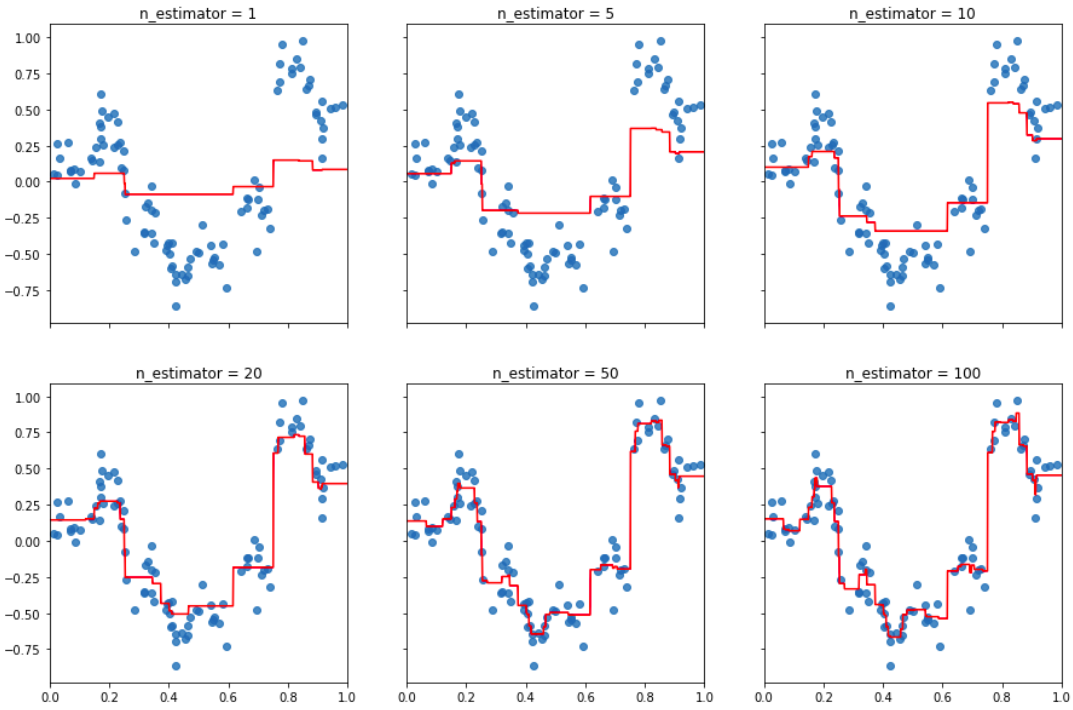

Figure 3: Gradient Boosting Regression Tree Output

```
        # Your code goes here

    def predict(self, test_data):
        '''
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''
        # Your code goes here
        test_predict = np.zeros(test_data.shape[0])
        for estimator in self.estimators:
            test_predict += self.learning_rate * estimator.predict(test_data)
        return test_predict
        # Your code goes here
```

Q6. Run the code provided to build gradient boosting models on the regression data sets krr-train.txt, and include the plots generated.

Fig. 3 is the required output. The relevant code follows.

```
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(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)
```

3 Image Classification with Gradient Boosting

Q7. Give the expression of the negative gradient step direction, or pseudo residual, $-g_m$ for the logistic loss as a function of the prediction function f_{m-1} at the previous iteration and the dataset points $(x_i, y_i)_{i=1}^n$. What is the dimension of g_m?

$l(m) = \ln(1 + e^{-m})$

$l(y_i, f_{m-1}(x_i)) = \ln(1 + e^{-y_i f_{m-1}(x_i)})$

For a single point, we can find the derivative w.r.t the function output at that point as:

$-g_{m_i} = -\frac{\partial\, l(y_i, f_{m-1}(x_i))}{\partial\, f_{m-1(x_i)}} = -\frac{\partial\, \ln(1+e^{-y_i f_{m-1}(x_i)})}{\partial\, f_{m-1(x_i)}} = \frac{y_i e^{-y_i f_{m-1}(x_i)}}{1+e^{-y_i f_{m-1}(x_i)}} = \frac{y_i}{1+e^{y_i f_{m-1}(x_i)}}$

For all points $(x_i, y_i)_{i=1}^n$: $-g_m = \left[\frac{y_1}{1+e^{y_1 f_{m-1}(x_1)}},\ \frac{y_2}{1+e^{y_2 f_{m-1}(x_2)}}\ \ldots\ \frac{y_n}{1+e^{y_n f_{m-1}(x_n)}}\right]$

This negative gradient $-g_m$ is in $\mathbb{R}^n$ i.e. one coordinate for each point in the dataset.

Q8. Write an expression for h_m as an argmin over functions h in $\mathcal{F}$

The goal of this h_m is to learn the projection of $-g_m$ in our hypothesis space i.e. we want the tree h that minimizes the negative gradient at each of the observed points.

$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left[\left(\frac{y_i}{1+e^{y_i f_{m-1}(x_i)}}\right) - h(x_i)\right]^2$

Q9. Plot the train and test accurary as a function of the number of estimators.

Fig. 4 is the required plot.

Relevant Code:

```
X_train, X_test, y_train, y_test = pre_process_mnist_01()
train_res = []
test_res = []
n_estimators = [2, 5, 10, 100, 200]
clfs = []
for n in n_estimators:
    print(n)
    clf = GradientBoostingClassifier(loss='deviance', max_depth=3,
        n_estimators = n)
    clf.fit(X_train, y_train)
    preds = clf.predict(X_train)
    accuracy = accuracy_score(y_train, preds)
    train_res.append(accuracy)
    preds = clf.predict(X_test)
    accuracy = accuracy_score(y_test, preds)
```

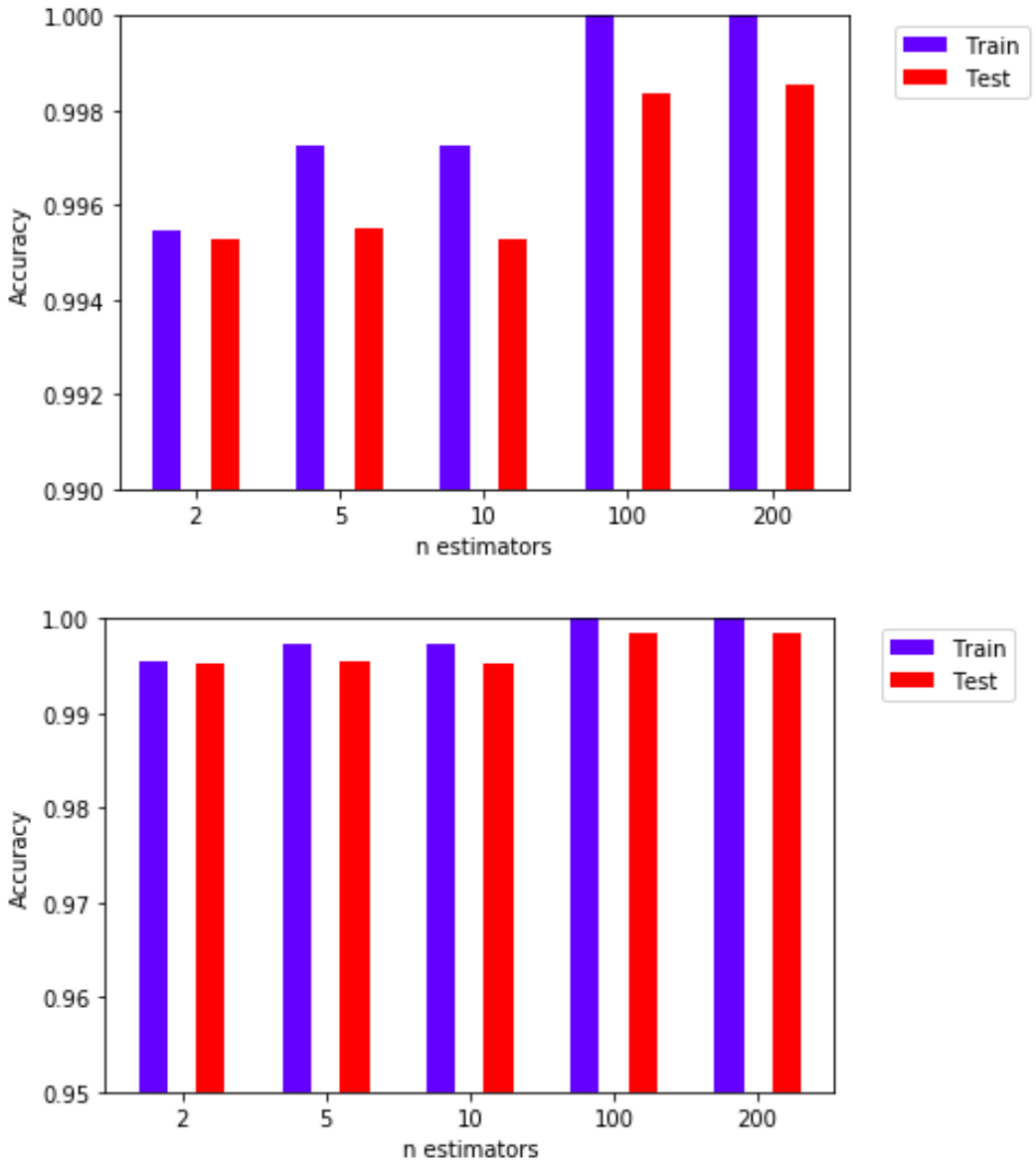

Figure 4: Train and test accuracy as a function of number of estimators used. Please note that the only difference between the two figures is the scale of the Y-axis. Both included for completeness.

```
    test_res.append(accuracy)
    clfs.append(clf)

plt.bar([i-0.2 for i in range(len(n_estimators))], train_res, width=0.2,
    label="Train", color='b')
plt.bar([i+0.2 for i in range(len(n_estimators))], test_res, width=0.2,
    label="Test", color='r')
plt.legend(bbox_to_anchor=(1.05, 1))
plt.ylim((0.99, 1))
plt.xlabel("n estimators")
plt.ylabel("Accuracy")
plt.xticks([i for i in range(len(n_estimators))], n_estimators)
plt.show()

plt.bar([i-0.2 for i in range(len(n_estimators))], train_res, width=0.2,
    label="Train", color='b')
plt.bar([i+0.2 for i in range(len(n_estimators))], test_res, width=0.2,
    label="Test", color='r')
plt.legend(bbox_to_anchor=(1.05, 1))
plt.ylim((0.95, 1))
plt.xlabel("n estimators")
plt.ylabel("Accuracy")
plt.xticks([i for i in range(len(n_estimators))], n_estimators)
plt.show()
```

4 Image Classification with Random Forests

Q10. Explain in your own words the construction principle of random forests

Random forest classifiers are an ensemble of decision trees. Each individual tree in the forest has a vote and the output of the ensemble is chosen by majority voting. The idea is that a set of models that are sufficiently uncorrelated will likely outperform any one model. To train the forest and ensure that the trees are all different, we can do bagging i.e. sampling from dataset with replacement, in the training process. So if there are N training examples, each tree sees N examples when it is trained but not necessarily the same subset. Additionally each tree is also given only a (random) subset of features that can be used to perform the splits. This ensures that the different trees give you different output and ensure that your majority vote is likely to work well.

Q11. Using the scikit learn implementation of RandomForestClassifier with the entropy loss (criterion='entropy') and trees of maximum depth 3, fit the preprocessed binary MNIST dataset with 2, 5, 10, 50, 100 and 200 estimators.

Fig. 5 is the output when varying the number of estimators.

Relevant Code for Q11:

```
from sklearn.ensemble import RandomForestClassifier

opt_train_res = []
opt_test_res = []
opt_n_estimators = [2, 5, 10, 50, 100, 200]
opt_clfs = []
for n in n_estimators:
    print(n)
    clf = RandomForestClassifier(criterion='entropy', max_depth=3,
        n_estimators = n)
    clf.fit(X_train, y_train)
    preds = clf.predict(X_train)
    accuracy = accuracy_score(y_train, preds)
    opt_train_res.append(accuracy)
    preds = clf.predict(X_test)
    accuracy = accuracy_score(y_test, preds)
    opt_test_res.append(accuracy)
    opt_clfs.append(clf)

plt.bar([i-0.2 for i in range(len(n_estimators))], opt_train_res, width=0.2,
    label="Train", color='b')
plt.bar([i+0.2 for i in range(len(n_estimators))], opt_test_res, width=0.2,
    label="Test", color='r')
```

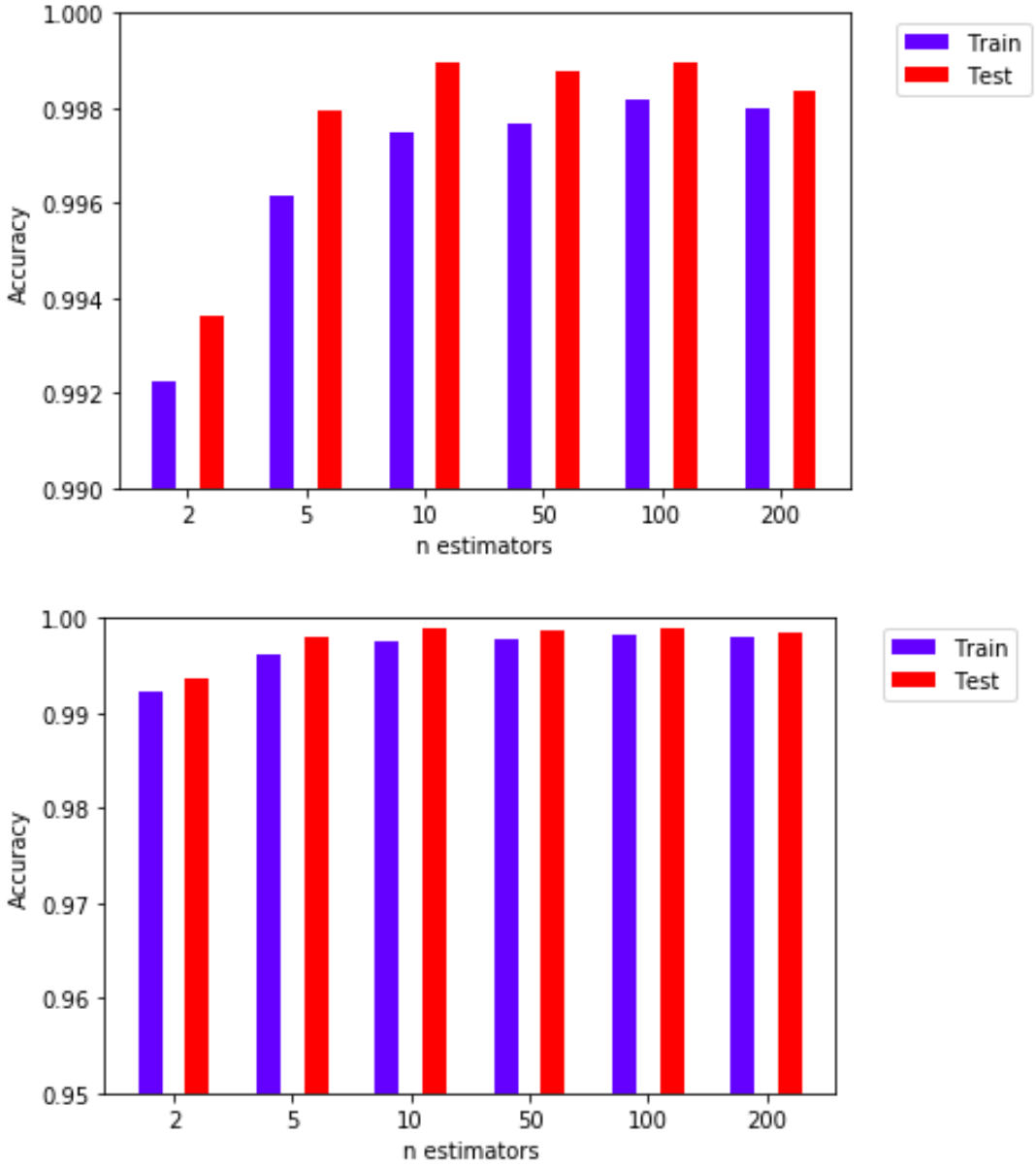

Figure 5: RandomForestClassifier Output when varying number of estimators. Please note that the only difference between the two figures is the scale of the Y-axis. Both included for completeness.

```
plt.legend(bbox_to_anchor=(1.05, 1))
plt.ylim((0.99, 1))
plt.xlabel("n estimators")
plt.ylabel("Accuracy")
plt.xticks([i for i in range(len(n_estimators))], n_estimators)
plt.show()

plt.bar([i-0.2 for i in range(len(n_estimators))], opt_train_res, width=0.2,
    label="Train", color='b')
plt.bar([i+0.2 for i in range(len(n_estimators))], opt_test_res, width=0.2,
    label="Test", color='r')
plt.legend(bbox_to_anchor=(1.05, 1))
plt.ylim((0.95, 1))
plt.xlabel("n estimators")
plt.ylabel("Accuracy")
plt.xticks([i for i in range(len(n_estimators))], n_estimators)
plt.show()
```

Q12. What general remark can you make on overfitting for Random Forests and Gradient Boosted Trees? Which method achieves the best train accuracy overall? Is this result expected? Can you think of a practical disadvantage of the best performing method? How do the algorithms compare in term of test accuracy?

For Gradient Boosting Trees, the test accuracy tends to be slightly lower than the train accuracy but since the difference is very small and both values are above 99%, it would be overstating it to say there is overfitting. This holds across all values of number of estimators but expectedly becomes more pronounced as we increase this number as there are more iterations in fitting the residual leading to near perfect train accuracy. For the Random Forests, the test accuracy tends to be higher than train accuracy across all values of number of estimators so there is no overfitting.

The best train accuracy is 1.0 or perfect accuracy for the Gradient Boosted trees at the parameters when 100 and 200 estimators are used. This is not unexpected since if you keep fitting the residual continuously and have a very large number of trees, eventually the train output will be near perfect.

The practical disadvantage is that as the number of estimators is so high, it takes time to train in practise and at inference time you would need the output from all these models before making your prediction. Given this overhead plus the fact that we see only marginal improvement increasing the number of estimators from 10 to 200, it might not be worth the additional improvement. Additionally we don't see it in this case much but overfitting becomes a concern if the number of estimators (or depth of trees) becomes too high.

The best test accuracy is obtained with the Random Forest Classifier with 10 estimators. On balance the test accuracy is marginally higher on Random Forest Classifier over the Gradient Boosted Tree with the corresponding number of estimators.