DS-GA 1003 - Homework 6

Eric Niblock

April 11, 2021

In this problem we'll implement decision trees for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the loss function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here. We'll work with the classification and regression data sets from previous assignments.

1. **Complete the `compute_entropy` and `compute_gini` functions.**

   The following functions compute the entropy and gini of our data,

```
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
    '''
    flat = label_array.flatten()
    uni = np.unique(flat)
    entropy = 0

    for i in uni:
        p = np.count_nonzero(flat == i)/len(flat)
        entropy += -1*p*np.log(p)

    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
```

```
        '''
        flat = label_array.flatten()
        uni = np.unique(flat)
        gini = 0

        for i in uni:
            p = np.count_nonzero(flat == i)/len(flat)
            gini += p*(1-p)

        return gini
```

2. **Complete the class `Decision_Tree`, given in the skeleton code. The intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable self.depth, with the root node having depth 0. The main job of the fit function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.**

   The following code is the completed decision tree class, with all associated methods,

```
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 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

    def fit(self, x, y):
```

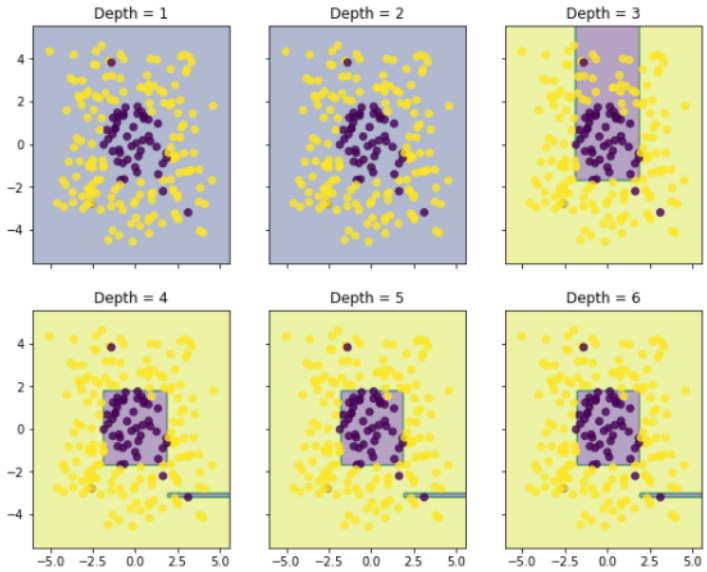
```
'''
This should fit the tree classifier by setting the values self.is_leaf,
self.split_id (the index of the feature we want ot split on, if we're splitting),
self.split_value (the corresponding value of that feature where the split is),
and self.value, which is the prediction value if the tree is 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
'''

if self.depth == self.max_depth:
    self.is_leaf = True
    self.value = self.leaf_value_estimator(y)
    return(self)
if len(y) < self.min_sample:
    self.is_leaf = True
    self.value = self.leaf_value_estimator(y)
    return(self)

self.find_best_feature_split(x, y)

if self.split_id == None:
    self.is_leaf = True
    self.value = self.leaf_value_estimator(y)
    return(self)
else:
    num_f = x.shape[1]
    X = np.concatenate([x,y],1)
    xleft = X[X[:,self.split_id]<=self.split_value][:,:num_f]
    xright = X[X[:,self.split_id]>self.split_value][:,:num_f]
    yleft = X[X[:,self.split_id]<=self.split_value][:,-1].reshape(-1,1)
    yright = X[X[:,self.split_id]>self.split_value][:,-1].reshape(-1,1)

    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(xleft,yleft)
    self.right.fit(xright,yright)

    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: a numpy array of training data, shape = (n_node)
    :param y: a numpy array of labels, shape = (n_node, 1)
    '''

    x_node = x_node[:,feature_id]

    zipped = zip(x_node, y_node)
    pairs = sorted(zipped)
    tups = zip(*pairs)
    x_node_sort, y_node_sort = [ list(tuple) for tuple in  tups]
    best_loss = self.split_loss_function(np.array(y_node_sort).reshape(-1,1))
    split_value = None

    for i in range(1,len(x_node_sort)):
        test_left_y = y_node_sort[:i]
        test_right_y = y_node_sort[i:]
        loss_l = (len(test_left_y)/len(y_node_sort))
            *self.split_loss_function(np.array(test_left_y).reshape(-1,1))
        loss_r = (len(test_right_y)/len(y_node_sort))
            *self.split_loss_function(np.array(test_right_y).reshape(-1,1))
        tot_loss = loss_l+loss_r
        if tot_loss < best_loss:
            best_loss = tot_loss
            split_value = (x_node_sort[i-1]+x_node_sort[i])/2

    return split_value, best_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.
    :param X: a numpy array of training data, shape = (n_node, 1)
    :param y: a numpy array of labels, shape = (n_node, 1)
    '''

    best_loss = self.split_loss_function(np.array(y_node).reshape(-1,1))
    self.split_id = None
    self.split_value = None


    for feature_id in range(x_node.shape[1]):

        split_value, loss = find_best_split(x_node, y_node, feature_id)

        if loss < best_loss:
```

```
                self.split_id = feature_id
                self.split_value = split_value
                best_loss = 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)
```

3. Run the code provided that builds trees for the two-dimensional classification data. Include the results. For debugging, you may want to compare results with sklearn's decision tree (code provided in the skeleton code). For visualization, you'll need to install `graphviz`.

Below are the results of running our decision tree class to fit our data. The results are consistent with the results provided by sklearn's decision tree,

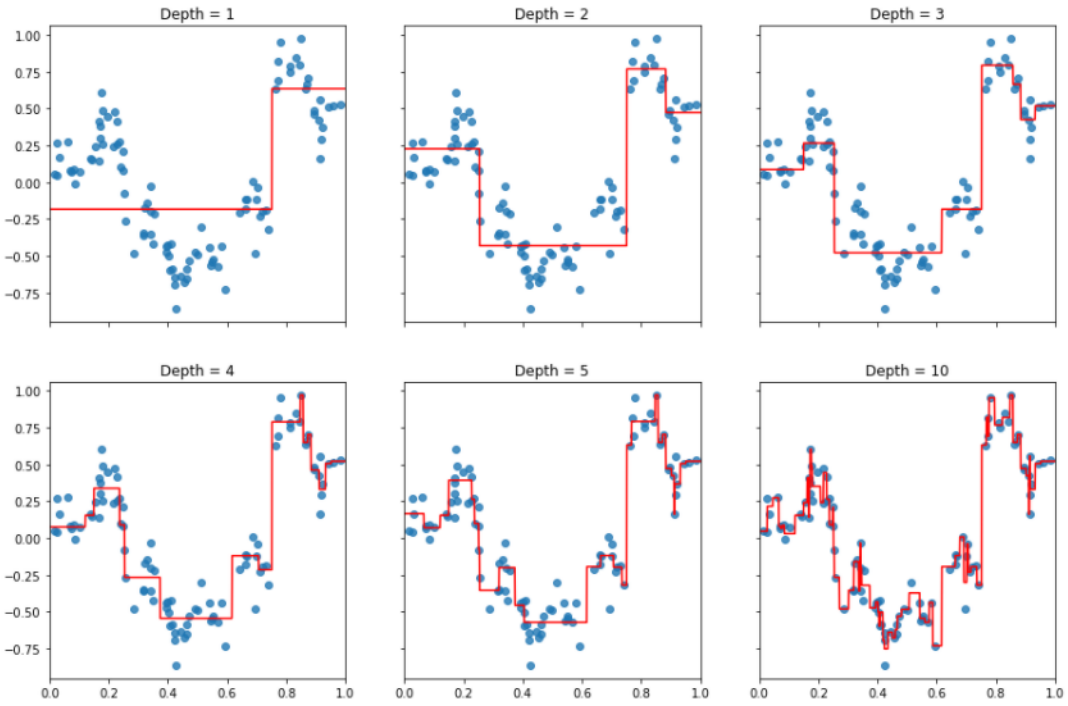


4. **Complete the function `mean_absolute_deviation_around_median` (MAE). Use the code provided to fit the `Regression_Tree` to the krr dataset using both the MAE loss and median predictions. Include the plots for the 6 fits.**

First, we provide the MAE function, which is given below,

```
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
    '''
    med = np.median(y)
    mae = np.mean(np.abs(y-med))
    return mae
```

Using our MAE function, we produced the following regression trees with various depths. We compared these results to sklearn's regressor, which provided sufficiently similar results,



Recall the general gradient boosting algorithm, for a given loss function ℓ and a hypothesis space $\mathcal{F}$ of regression functions (i.e. functions mapping from the input space to $\mathbb{R}$):

0: Initialize $f_0(x) = 0$.

1: For $m = 1$ to M:

(a) Compute:

$$\mathbf{g}_m = \left(\frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i)\right) \right)_{j=1}^{n}$$

(b) Fit regression model to $-\mathbf{g}_m$:

$$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left(\left(-\mathbf{g}_m\right)_i - h(x_i) \right)^2 .$$

(c) Choose fixed step size $\nu_m = \nu \in (0, 1]$, or take

$$\nu_m = \arg\min_{\nu > 0} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + \nu h_m(x_i)\right) .$$

(d) Take the step:

$$f_m(x) = f_{m-1}(x) + \nu_m h_m(x)$$

3: Return f_M.

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. One of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function.

First we'll keep things simple and consider the standard regression setting with square loss. In this case the we have $\mathcal{Y} = \mathbb{R}$, our loss function is given by $\ell(\hat{y}, y) = \frac{1}{2}\left(\hat{y} - y\right)^2$, and at the m'th round of gradient boosting, we have

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

5. Complete the `gradient_boosting` class. As the base regression algorithm to compute the argmin, you should use sklearn's regression tree. You should use the square loss for the tree splitting rule (`criterion` keyword argument) and use the default sklearn leaf prediction rule from the `predict` method [1]. We will also use a constant step size ν.

Below is out completed implementation of the gradient boosting class,

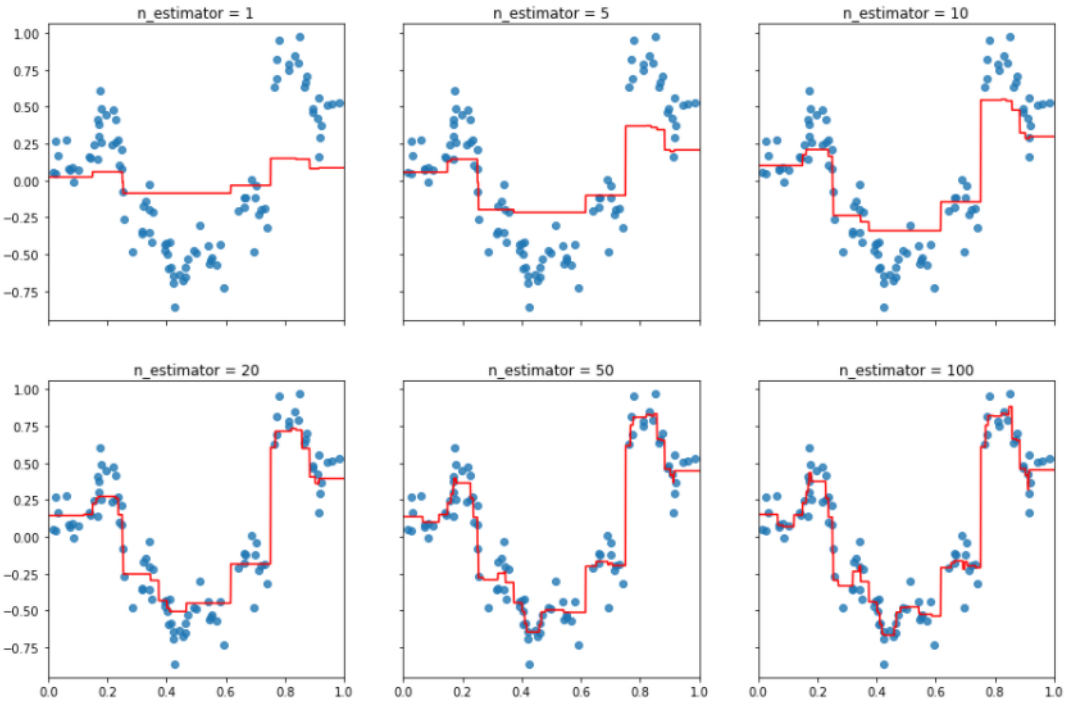
[1] Examples of usage are given in the skeleton code to debug previous problems, and you can check the docs `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html`

```
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.f0 = DecisionTreeRegressor(max_depth=self.max_depth,\
                                        min_samples_leaf=self.min_sample)
        self.f0.fit(train_data,train_target)
        pred = self.learning_rate*self.f0.predict(train_data)
        res = self.pseudo_residual_func(train_target.reshape(-1),pred)
        h_m = DecisionTreeRegressor(max_depth=self.max_depth,\
                                    min_samples_leaf=self.min_sample)
        h_m.fit(train_data,res)
        self.estimators.append(h_m)

        for e in range(self.n_estimator - 1):
            pred = self.learning_rate*self.f0.predict(train_data)
            for model in self.estimators:
                pred += self.learning_rate*model.predict(train_data)
            res = self.pseudo_residual_func(train_target.reshape(-1),pred)
            h_m = DecisionTreeRegressor(max_depth=self.max_depth,\
                                        min_samples_leaf=self.min_sample)
            h_m.fit(train_data,res)
```

```
            self.estimators.append(h_m)

    def predict(self, test_data):
        '''
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''
        test_predict = self.learning_rate*self.f0.predict(test_data)
        for model in range(len(self.estimators)):
            test_predict += self.learning_rate*\
            self.estimators[model].predict(test_data)

        return test_predict
```

6. **Run the code provided to build gradient boosting models on the regression data sets `krr-train.txt`, and include the plots generated. For debugging you can use the sklearn implementation of `GradientBoostingRegressor`[2].**

   Below are the results of our gradient boosting models, which yields similar results to that of the sklearn implementation,



[2] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

In this problem we will consider the classification of MNIST, the dataset of handwritten digits images, with ensembles of trees. For simplicity, we only retain the '0' and '1' examples and perform binary classification.

First we'll derive a special case of the general gradient boosting framework: BinomialBoost. Let's consider the classification framework, where $\mathcal{Y} = \{-1, 1\}$. In lecture, we noted that AdaBoost corresponds to forward stagewise additive modeling with the exponential loss, and that the exponential loss is not very robust to outliers (i.e. outliers can have a large effect on the final prediction function). Instead, let's consider the logistic loss

$$\ell(m) = \ln\left(1 + e^{-m}\right),$$

where $m = yf(x)$ is the margin.

7. **Give the expression of the negative gradient step direction, or pseudo residual, $-\mathbf{g}_m$ 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?**

   We find the following negative gradient,

$$-\frac{\partial \ell(yf(x))}{\partial f(x)} = -\frac{\partial}{\partial f(x)}\left(\ln(1 + e^{-yf(x)})\right) = \frac{ye^{-yf(x)}}{1 + e^{-yf(x)}} \tag{1}$$

   And therefore, in terms of f_{m-1} we have,

$$-g_m = \frac{ye^{-yf_{m-1}(x)}}{1 + e^{-yf_{m-1}(x)}} \tag{2}$$

   With g_m being of dimension n.

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

   Below we have the expression for h_m,

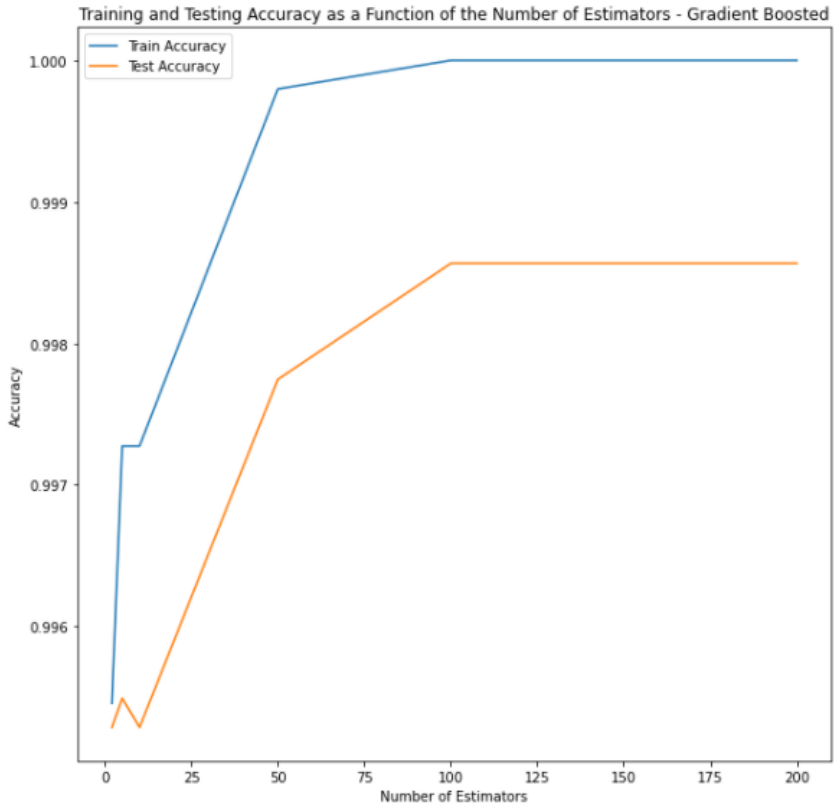
$$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left((-\mathbf{g}_m)_i - h(x_i)\right)^2 = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left(\frac{ye^{-yf_{m-1}(x_i)}}{1 + e^{-yf_{m-1}(x_i)}} - h(x_i)\right)^2 \tag{3}$$

9. **Load the MNIST dataset using the helper preprocessing function in the skeleton code. Using the scikit learn implementation of `GradientBoostingClassifier`, with the logistic loss (`loss='deviance'`) and trees of maximum depth 3, fit the data with 2, 5, 10, 100 and 200 iterations (estimators). Plot the train and test accurary as a function of the number of estimators.**

Below we include the code associated with running sklearn's gradient boosting classifier given different numbers of estimators,

```
train_acc = []
test_acc = []
for es in [2,5,10,50,100,200]:
    clf = GradientBoostingClassifier(loss='deviance', max_depth=3, n_estimators=es)
    clf.fit(X_train, y_train)
    y_pred_train = clf.predict(X_train)
    y_pred_test = clf.predict(X_test)
    train_acc.append(skl.metrics.accuracy_score(y_train, y_pred_train))
    test_acc.append(skl.metrics.accuracy_score(y_test, y_pred_test))
```

Furthermore, we have the following results concerning our training and testing accuracy,



10. Another type of ensembling method we discussed in class are random forests. Explain in your own words the construction principle of random forests.

Random forests are ensemble models composed of many, shallow and 'weak' decision trees. The underlying principle is that a plethora of relatively uncorrelated decision trees preforms better than one complex and intricate model. This is because the panel of decision trees which votes on a single example will return a verdict with lower variance than that of the individual model. This makes it more difficult for random forests to succumb to overfitting, which is common concerning decision trees.

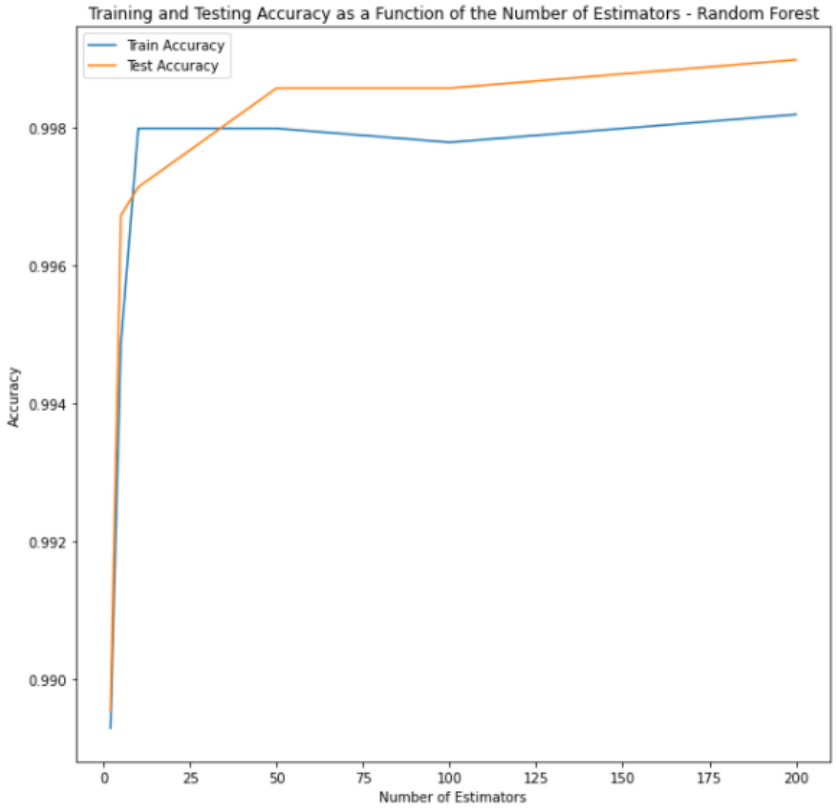
11. Using the scikit learn implementation of `RandomForestClassifier`[3], with the entropy loss (`criterion='entropy'`) and trees of maximum depth 3, fit the preprocessed binary MNIST dataset with 2, 5, 10, 50, 100 and 200 estimators.

Below we include the code associated with running sklearn's random forest classifier given different numbers of estimators,

```
train_acc = []
test_acc = []
for es in [2,5,10,50,100,200]:
    clf = RandomForestClassifier(criterion='entropy', max_depth=3, n_estimators=es)
    clf.fit(X_train, y_train)
    y_pred_train = clf.predict(X_train)
    y_pred_test = clf.predict(X_test)
    train_acc_0.append(skl.metrics.accuracy_score(y_train, y_pred_train))
    test_acc_0.append(skl.metrics.accuracy_score(y_test, y_pred_test))
```

Furthermore, we have the following results concerning our training and testing accuracy,

[3] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier



12. 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?

In general, gradient boosting is prone to overfitting whereas random forests are resistant to overfitting. However, concerning the previous implementations of each model, overfitting did not appear to be an issue for either.

Our gradient boosted model achieved the best training accuracy (100%), which is to be expected considering gradient boosted models are prone to overfitting.

In the end, the random forest produced slightly better test accuracy at 99.897%. The main practical disadvantage of a random forest (which is the best performing model here) is that the number of decision trees used is often large, thus slowing down real time predictions.