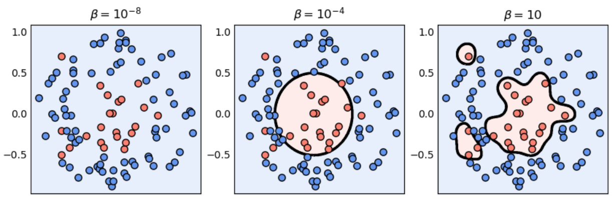
12.7 Two-class classification with the RBF kernel

Implement the RBF kernel in Example 12.9 and perform nonlinear two-class classification on the dataset shown in the middle row of Figure 12.3 using $\beta = 10^{-8}$, $\beta = 10^{-4}$, and $\beta = 10$. For each case produce a misclassification history plot to show that your results match what is shown in the figure.

△ Each model was trained for 10 iterations.

Cost function: Softmax

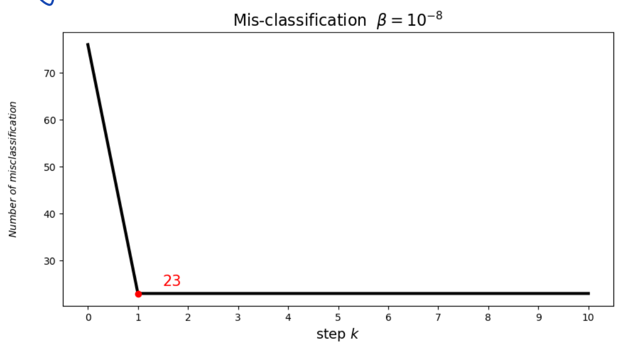
Optimization method: Newtons_method (epsilon $= 10^{-10}$)

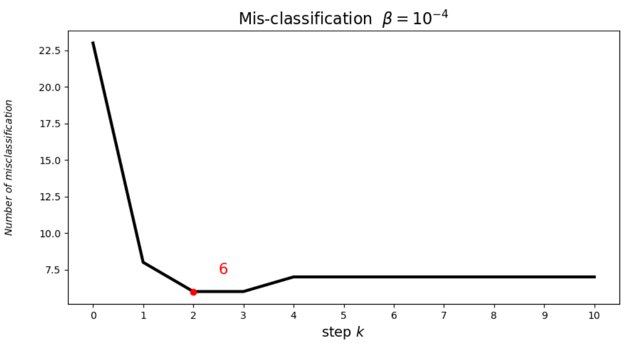


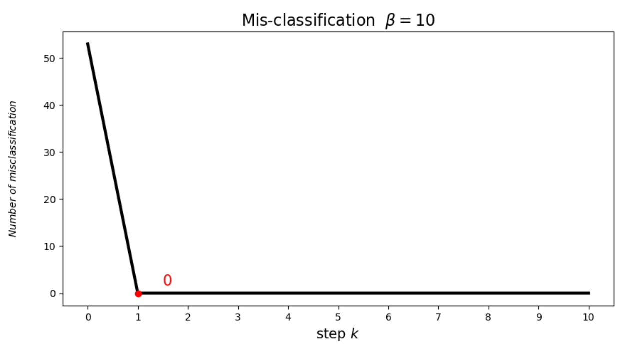
For each case, we have produced its mis-classification history as below.

* The red dot corresponds to the point with the minimum cost in cost-history (the optimal model)

* The red number represents the number of wrong predictions for the corres-ponding model.







Example 12.9 The Radial Basis Function (RBF) kernel

Another popular choice of kernel is the Radial Basis Function (RBF) kernel defined entry-wise over the input data as

$$h_{i,j} = e^{-\beta \left\| \mathbf{x}_i - \mathbf{x}_j \right\|_2^2} \tag{12.36}$$

where $\beta > 0$ is a *hyperparameter* that must be tuned to the data. While the RBF kernel is typically defined directly as the kernel matrix in Equation (12.36), it can be traced back to an explicit feature transformation as with the polynomial and Fourier kernels. That is, we can find the explicit form of the fixed-shape feature transformation $\mathbf{f}$ such that

$$h_{i,j} = \mathbf{f}_i^T \mathbf{f}_j \tag{12.37}$$

where $\mathbf{f}_i$ and $\mathbf{f}_j$ are the feature transformations of the input points $\mathbf{x}_i$ and $\mathbf{x}_j$, respectively. The RBF feature transformation is different from polynomial and Fourier transformations in that its associated feature vector $\mathbf{f}$ is *infinite-dimensional*. For example, when $N = 1$ the feature vector $\mathbf{f}$ takes the form

$$\mathbf{f} = \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \\ \vdots \end{bmatrix} \tag{12.38}$$

where the mth entry (or feature) is defined as

$$f_m(x) = e^{-\beta x^2} \sqrt{\frac{(2\beta)^{m-1}}{(m-1)!}}\, x^{m-1} \quad \text{for all } m \geq 1. \tag{12.39}$$

When $N > 1$ the corresponding feature vector takes on an analogous form which is also infinite in length, making it impossible to even construct and store such a feature vector (regardless of the input dimension).

Notice that the shape (and hence fitting behavior) of RBF kernels depends on the setting of their hyperparameter β. In general, the larger β is set the more complex an associated model employing an RBF kernel becomes. To illustrate this, in Figure 12.3 we show three examples of supervised learning: regression (top row), two-class classification (middle row), and multi-class classification (bottom row), using the RBF kernel with three distinct settings of β in each instance. This creates underfitting (left column), reasonable predictive behavior (middle column), and overfitting behavior (right column). In each instance Newton's method was used to minimize each corresponding cost, and consequently tune each model's parameters. In practice β is set via *cross-validation* (see, e.g., Example 12.10).

● 12-7 Two-class classification with the RBF kernel

```
from matplotlib import pyplot as plt, gridspec
from mlrefined_libraries.nonlinear_superlearn_library.kernel_visualizer import Visualizer
from mlrefined_libraries.nonlinear_superlearn_library.kernel_lib.kernels import Setup as K_setup
import autograd.numpy as np
from mlrefined_libraries.math_optimization_library import static_plotter
import copy
from autograd import value_and_grad
from autograd import hessian
from autograd.misc.flatten import flatten_func

plotter = static_plotter.Visualizer()



class two_class_with_RBF_kernel(object):
   def __init__(self, file_path):
      self.models = []
      self.train_cost_histories = []
      self.weight_histories = []
      data = np.loadtxt(file_path, delimiter=',')
      self.x = copy.deepcopy(data[:-1, :])
      self.y = copy.deepcopy(data[-1:, :])

   def decent_ini(self):
      self.w0 = 0.05 * np.random.randn(np.size(self.y) + 1, 1)

   def normalization(self):
      x_means = np.nanmean(self.x, axis=1)[:, np.newaxis]
      x_stds = np.nanstd(self.x, axis=1)[:, np.newaxis]
      ind = np.argwhere(x_stds < 10 ** (-2))
      if len(ind) > 0:
         ind = [v[0] for v in ind]
         adjust = np.zeros((x_stds.shape))
         adjust[ind] = 1.0
         x_stds += adjust
      self.normalizer = lambda data: (data - x_means) / x_stds
      self.x = self.normalizer(self.x)

   def kernel(self, name, **kwargs):
      self.transformer = K_setup(name, **kwargs)
      self.H_train = self.transformer.kernel(self.x_train, self.x_train)
      self.H = lambda x: self.transformer.kernel(self.x_train, x)

   def add_kernel_to_cost(self):
```

```
        self.train_cost = lambda w, iter: self.softmax(w, self.H_train, self.y_train, iter)
        self.model = lambda x, w: w[0] + np.dot(self.H(x), w[1:])

    def linear_model(self, f, w):
        a = w[0] + np.dot(f.T, w[1:])
        return a.T

    def counting_mis_classification(self, w, x, y):
        y_predict = np.sign(self.linear_model(x, w))
        num_misclass = len(np.argwhere(y != y_predict))
        return num_misclass

    def softmax(self, w, H, y, iter):
        f_p = H[:, iter]
        y_p = y[:, iter]
        cost = np.sum(np.log(1 + np.exp(-y_p * self.linear_model(f_p, w))))
        return cost / float(np.size(y_p))

    def dataset_split(self, train_portion):
        shuffled_data = np.random.permutation(self.x.shape[1])
        train_num = int(np.round(train_portion * len(shuffled_data)))
        self.train_inds = shuffled_data[:train_num]
        self.valid_inds = shuffled_data[train_num:]
        self.x_train = self.x[:, self.train_inds]
        self.x_valid = self.x[:, self.valid_inds]
        self.y_train = self.y[:, self.train_inds]
        self.y_valid = self.y[:, self.valid_inds]

    def newtons_method(self, g, max_its, w, num_pts, batch_size, epsilon):
        g_flat, unflatten, w = flatten_func(g, w)
        gradient = value_and_grad(g_flat)
        hess = hessian(g_flat)
        train_hist = [g_flat(w, np.arange(num_pts))]
        w_hist = [unflatten(w)]
        num_batches = int(np.ceil(np.divide(num_pts, batch_size)))
        for k in range(max_its):
            print('running iteration:' + str(k + 1) + ' of ' + str(max_its))
            for b in range(num_batches):
                batch_inds = np.arange(b * batch_size, min((b + 1) * batch_size, num_pts))
                cost_eval, grad_eval = gradient(w, batch_inds)
                hess_eval = hess(w, batch_inds)
                hess_eval.shape = (int((np.size(hess_eval)) ** 0.5), int((np.size(hess_eval)) ** 0.5))
                A = hess_eval + epsilon * np.eye(np.size(w))
                b = grad_eval
```

```
                w = np.linalg.lstsq(A, np.dot(A, w) - b)[0]
            w_hist.append(unflatten(w))
            train_hist.append(g_flat(w, np.arange(num_pts)))
        return w_hist, train_hist

    def train(self, max_its, epsilon, beta):
        self.mis_class = []
        self.normalization()
        self.dataset_split(1)
        self.decent_ini()
        self.add_kernel_to_cost()
        self.kernel(name='gaussian', beta=beta, scale=0)
        self.num_pts = np.size(self.y_train)
        self.batch_size = np.size(self.y_train)
        w_his, c_his = self.newtons_method(self.train_cost, max_its, self.w0, self.num_pts,
self.batch_size, epsilon)
        self.weight_histories.append(w_his)
        self.train_cost_histories.append(c_his)
        for w in w_his:
            self.mis_class.append(self.counting_mis_classification(w, self.H_train, self.y_train))
        self.models.append(copy.deepcopy(self))

def plot_mismatching_histories(histories, start, title='', **kwargs):
    colors = ['black', 'aqua', 'magenta', 'k', 'chocolate']
    fig = plt.figure(figsize=(10, 5))
    gs = gridspec.GridSpec(1, 1)
    ax = plt.subplot(gs[0])
    labels = [' ', ' ', ' ']
    if 'labels' in kwargs:
        labels = kwargs['labels']
    points = False
    if 'points' in kwargs:
        points = kwargs['points']
    for c in range(len(histories)):
        history = histories[c]
        label = 0
        if c == 0:
            label = labels[0]
        elif c == 1:
            label = labels[1]
        else:
            label = labels[2]
        x_axis = np.arange(start, len(history), 1)
        ind = np.argmin(history)
```

```
        plt.scatter(x_axis[ind], history[ind], color='r', zorder=2)
        plt.text(x_axis[ind] + 0.5, history[ind] + 1, '%.0f' % history[ind], fontsize=15, ha='left',
va='bottom',
                 color='r')
        if np.size(label) == 0:
            ax.plot(x_axis, history[start:], linewidth=3 * 0.8 ** c, color=colors[c], zorder=1)
        else:
            ax.plot(x_axis, history[start:], linewidth=3 * 0.8 ** c, color=colors[c],
                    label=label, zorder=1)
        if points:
            ax.scatter(np.arange(start, len(history), 1), history[start:], s=90, color=colors[c],
edgecolor='w',
                       linewidth=2, zorder=3)
    xlabel = 'step $k$'
    if 'xlabel' in kwargs:
        xlabel = kwargs['xlabel']
    ylabel = '$Number\ of\ misclassification$'
    if 'ylabel' in kwargs:
        ylabel = kwargs['ylabel']
    ax.set_xlabel(xlabel, fontsize=14)
    ax.set_ylabel(ylabel, fontsize=10, rotation=90, labelpad=25)
    plt.xticks(range(0, len(history), int(len(history) / 10)))
    ax.set_xlim([start - 0.5, len(history) - 0.5])
    plt.title(title, fontsize=16)
    plt.show()


if __name__ == "__main__":
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/new_circle_data.csv'
    betas = [10 ** (-8), 10 ** (-4), 10 ** 1]
    label = [r'$\beta =10^{-8}$', r'$\beta =10^{-4}$', r'$\beta =10$']
    models = []
    mis_his = []
    ind = 0
    for beta in betas:
        RBF = two_class_with_RBF_kernel(file_path)
        RBF.train(max_its=10, epsilon=10 ** (-10), beta=beta)
        models.append(copy.deepcopy(RBF))
        mis_his.append(RBF.mis_class)
        plot_mismatching_histories(histories=[RBF.mis_class], start=0, title='Mis-classification  ' +
str(label[ind]))
        ind += 1
    result_vis = Visualizer(file_path)
    result_vis.show_twoclass_runs(models, labels=label)
```

12.10 **An infinite-dimensional feature transformation**

Verify that the infinite-dimensional feature transformation defined in Equation (12.39) indeed yields the entry-wise form of the RBF kernel in Equation (12.36).

$$h_{i,j} = e^{-\beta\|\mathbf{x}_i - \mathbf{x}_j\|_2^2} \quad \Rightarrow \quad f_m(x) = e^{-\beta x^2}\sqrt{\frac{(2\beta)^{m-1}}{(m-1)!}}\,x^{m-1} \quad \text{for all } m \geq 1.$$

$$h_{ij} = f_i^T f_j = f_1(x_i)f_1(x_j) + f_2(x_i)f_2(x_j) + f_3(x_i)f_3(x_j) + \cdots$$

$$= \left(e^{-\beta x_i^2}\sqrt{\frac{(2\beta)^0}{0!}}\,x_i^0 \cdot e^{-\beta x_j^2}\sqrt{\frac{(2\beta)^0}{0!}}\,x_j^0\right)$$

$$+ \left(e^{-\beta x_i^2}\sqrt{\frac{(2\beta)^1}{1!}}\,x_j^1 \cdot e^{-\beta x_j^2}\sqrt{\frac{(2\beta)^1}{1!}}\,x_j^1\right)$$

$$+ \left(e^{-\beta x_i^2}\sqrt{\frac{(2\beta)^2}{2!}}\,x_j^2 \cdot e^{-\beta x_j^2}\sqrt{\frac{(2\beta)^2}{2!}}\,x_j^2\right)$$

$$= e^{-\beta x_i^2} \cdot e^{-\beta x_j^2}\left(\frac{(2\beta)^0}{0!}x_i^0 x_j^0 + \frac{(2\beta)^1}{1!}x_i x_j + \frac{(2\beta)^2}{2!}x_i^2 x_j^2\right)$$

According to Taylor series $e^x = \sum_{n=0}^{\infty}\frac{x^n}{n!}$ $\quad x = (2\beta x_i x_j)^m$

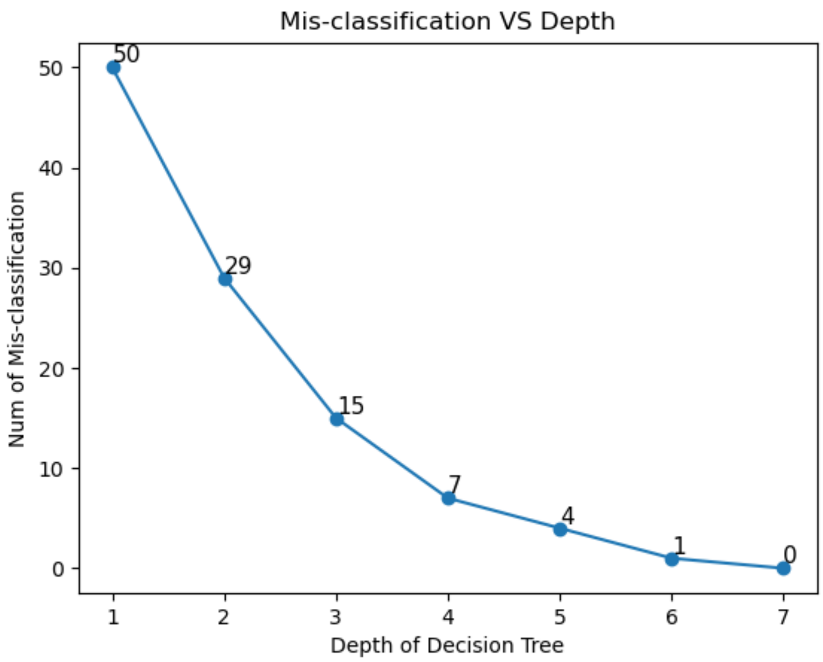
$$= e^{-\beta x_i^2} e^{-\beta x_j^2} e^{2\beta x_i x_j} = e^{-\beta(x_i^2 + x_j^2 + 2x_i x_j)} = e^{-\beta\|x_i - x_j\|_2^2}$$

14.4 **Code up a two-class classification tree**
Repeat the first experiment described in Example 14.4 by coding up a recursively defined two-class classification tree. You need not reproduce Figure 14.11. Instead, measure and plot the number of misclassifications at each depth of your tree.

Example 14.4 **Growing two maximum-depth classification trees**
In Figure 14.11 we illustrate the growth of a tree to a maximum depth of seven on a two-class classification dataset. In Figure 14.12 we do the same for a multi-class classification dataset with $C = 3$ classes. In both cases as the tree grows note how many parts of the input space do not change as leaves on the deeper branches become *pure*. By the time we reach a maximum depth of seven we have considerably overfit both datasets.

Sol: The figure below shows the trend of the number of misclassification in the decision tree as the depth increases.

In general, with the increase of the depth of the decision tree, the classification ability of the model will increase, but when the depth reaches a specific threshold, the over-fitting will occur.



The black numbers in the line chart represent the number of misclassification.

- ## 14-4 Code up a two-class classification tree

  - ### Part 1 Basic structure of tree

```
from autograd import numpy as np
import copy

left_his = []
right_his = []


class Stump:
   def __init__(self, x, y):
      self.x = x
      self.y = y
      self.make_stump()

   def counter(self, step, x, y):
      y_hat = step(x)[np.newaxis, :]
      vals, counts = np.unique(y, return_counts=True)
      balanced = 0
      for i in range(len(vals)):
         v = vals[i]
         c = counts[i]
         ind = np.argwhere(y == v)
         miss_val = 1
         if ind.size > 0:
            ind = [a[1] for a in ind]
            miss = np.argwhere(y_hat[:, ind] != y[:, ind])
            if miss.size > 0:
               miss = len([a[1] for a in miss])
               miss_val = (1 - miss / c)
         balanced += miss_val
      balanced = balanced / len(vals)
      return balanced

   def make_stump(self):
      # important constants: dimension of input N and total number of points P
      N = np.shape(self.x)[0]
      P = np.size(self.y)
      acc_matrix_right = [0] * N
      acc_matrix_left = [0] * N
      best_split = np.inf
```

```python
best_dim = np.inf
best_val = -np.inf
best_left_leaf = []
best_right_leaf = []
best_left_ave = []
best_right_ave = []
best_step = []
c_vals, c_counts = np.unique(self.y, return_counts=True)
self.c_counts = c_counts

for n in range(N):
    x_n = copy.deepcopy(self.x[n, :])
    y_n = copy.deepcopy(self.y)

    sorted_inds = np.argsort(x_n, axis=0)
    x_n = x_n[sorted_inds]
    y_n = y_n[:, sorted_inds]
    for p in range(P - 1):
        if y_n[:, p] != y_n[:, p + 1] and x_n[p] != x_n[p + 1]:
            # compute split point
            split = (x_n[p] + x_n[p + 1]) / float(2)
            y_n_left = y_n[:, :p + 1]
            y_n_right = y_n[:, p + 1:]
            c_left_vals, c_left_counts = np.unique(y_n_left, return_counts=True)
            c_right_vals, c_right_counts = np.unique(y_n_right, return_counts=True)

            prop_left = []
            prop_right = []
            for i in range(np.size(c_vals)):
                val = c_vals[i]
                count = c_counts[i]

                val_ind = np.argwhere(c_left_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_left_counts[val_ind][0][0]
                prop_left.append(val_count / count)

                # check right side
                val_ind = np.argwhere(c_right_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_right_counts[val_ind][0][0]
                prop_right.append(val_count / count)
```

```
            # array it
            prop_left = np.array(prop_left)
            best_left = np.argmax(prop_left)
            left_ave = c_vals[best_left]
            best_acc_left = prop_left[best_left]
            # left = y_n_left.size / y_n.size

            prop_right = np.array(prop_right)
            best_right = np.argmax(prop_right)
            right_ave = c_vals[best_right]
            best_acc_right = prop_right[best_right]
            # right = y_n_right.size / y_n.size
            val = (best_acc_left + best_acc_right) / 2

            # define leaves
            left_leaf = lambda x, left_ave=left_ave, dim=n: np.array([left_ave for v in x[dim, :]])
            right_leaf = lambda x, right_ave=right_ave, dim=n: np.array([right_ave for v in
x[dim, :]])

            # create stump
            step = lambda x, split=split, left_ave=left_ave, right_ave=right_ave, dim=n: np.array(
                [(left_ave if v <= split else right_ave) for v in x[dim, :]])

            # compute cost value on step
            # val = self.counter(step,self.x,self.y)

            if val > best_val:
                acc_matrix_right = prop_right
                acc_matrix_left = prop_left
                best_left_leaf = copy.deepcopy(left_leaf)
                best_right_leaf = copy.deepcopy(right_leaf)

                best_dim = copy.deepcopy(n)
                best_split = copy.deepcopy(split)
                best_val = copy.deepcopy(val)
                best_left_ave = copy.deepcopy(left_ave)
                best_right_ave = copy.deepcopy(right_ave)
                best_step = copy.deepcopy(step)

        # define globals
        self.step = best_step
        self.left_leaf = best_left_leaf
        self.right_leaf = best_right_leaf
```

```
        self.dim = best_dim
        self.split = best_split

        # sort x_n and y_n according to ascending order in x_n
        sorted_inds = np.argsort(self.x[best_dim, :], axis=0)
        self.x = self.x[:, sorted_inds]
        self.y = self.y[:, sorted_inds]

        # cull out points on each leaf
        left_inds = np.argwhere(self.x[best_dim, :] <= best_split).flatten()
        right_inds = np.argwhere(self.x[best_dim, :] > best_split).flatten()

        self.left_x = self.x[:, left_inds]
        self.right_x = self.x[:, right_inds]
        self.left_y = self.y[:, left_inds]
        self.right_y = self.y[:, right_inds]
        self.number_mis_class_left = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[0]
        self.number_mis_class_right = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[1]
        right_his.append(self.number_mis_class_right)
        left_his.append(self.number_mis_class_left)

    def caculate_mis_class(self, prop_right, prop_left):
        leaf_label_ind_left = np.argmax(prop_left)
        left_label_count = self.c_counts[leaf_label_ind_left]
        leaf_label_ind_right = np.argmax(prop_right)
        right_label_count = self.c_counts[leaf_label_ind_right]
        mis_class_left = self.left_x.shape[1] - round(left_label_count * prop_left[leaf_label_ind_left])
        mis_class_right = self.right_x.shape[1] - round(right_label_count *
prop_right[leaf_label_ind_right])
        return mis_class_left, mis_class_right
```

➢ Part 2 Build Tree

```
from matplotlib import pyplot as plt
from mlrefined_libraries.nonlinear_superlearn_library.recursive_tree_lib.ClassificationTree import
ClassificationStump
import copy
import autograd.numpy as np

depth_count = 1


class Decision_Tree(object):
```

```python
def __init__(self, file_path, depth):
    data = np.loadtxt(file_path, delimiter=',')
    self.misclassification = []
    x = data[:-1, :]
    y = data[-1:, :]
    self.depth = depth
    self.tree = Tree()
    stump = ClassificationStump.Stump(x, y)
    self.build_tree(stump, self.tree, depth)

def build_subtree(self, stump):
    best_split = stump.split
    best_dim = stump.dim
    left_x = stump.left_x
    right_x = stump.right_x
    left_y = stump.left_y
    right_y = stump.right_y
    left_stump = stump
    right_stump = stump
    if np.size(np.unique(left_y)) > 1:
        left_stump = ClassificationStump.Stump(left_x, left_y)
    else:
        right_stump.right_y = right_stump.left_y
        left_stump.number_mis_class_left = 0
        left_stump.number_mis_class_right = 0
    if np.size(np.unique(right_y)) > 1:
        right_stump = ClassificationStump.Stump(right_x, right_y)
    else:
        right_stump.left_y = right_stump.right_y
        right_stump.number_mis_class_left = 0
        right_stump.number_mis_class_right = 0
    return left_stump, right_stump

def build_tree(self, stump, node, depth):
    if depth > 1:
        node.split = stump.split
        node.dim = stump.dim
        node.left_leaf = stump.left_leaf
        node.right_leaf = stump.right_leaf
        node.step = stump.step
        node.number_mis_class_left = stump.number_mis_class_left
        node.number_mis_class_right = stump.number_mis_class_right
        left_stump, right_stump = self.build_subtree(stump)
        depth -= 1
```

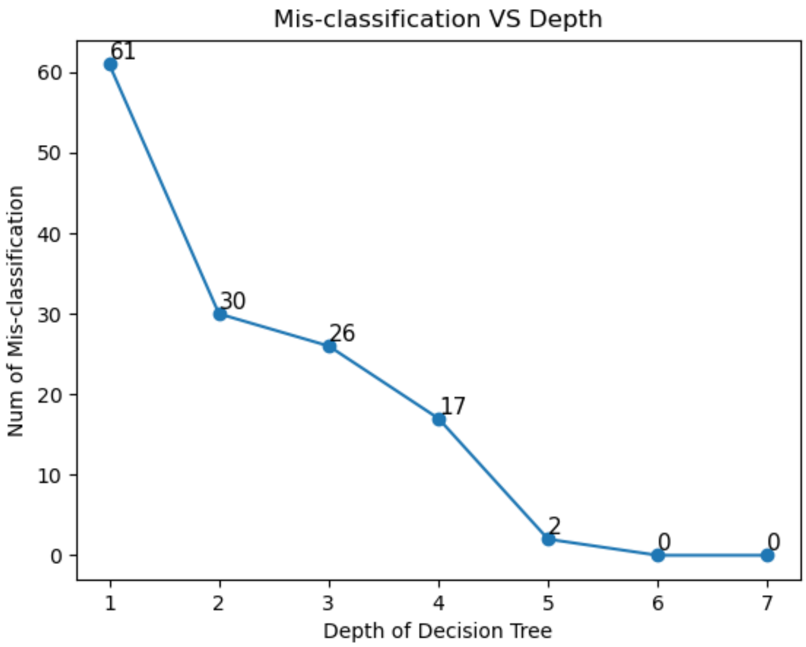
```python
            if left_stump.number_mis_class_right + left_stump.number_mis_class_left == 0 \
                    and right_stump.number_mis_class_right + right_stump.number_mis_class_left == 0:
                depth = 1
            node.left = Tree()
            node.right = Tree()
            return self.build_tree(right_stump, node.right, depth), self.build_tree(left_stump,
node.left, depth)
        else:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
            node.right_leaf = stump.right_leaf
            node.step = stump.step
            node.number_mis_class_left = stump.number_mis_class_left
            node.number_mis_class_right = stump.number_mis_class_right
            # node.all_miss = stump.all_miss
            self.misclassification.append(node.number_mis_class_left)
            self.misclassification.append(node.number_mis_class_right)

    # tree evaluator
    def evaluate_tree(self, val, depth):
        if depth > self.depth:
            return ('desired depth greater than depth of tree')

        tree = copy.deepcopy(self.tree)
        d = 0
        while d < depth:
            split = tree.split
            dim = tree.dim
            if val[dim, :] <= split:
                tree = tree.left
            else:
                tree = tree.right
            d += 1
        split = tree.split
        dim = tree.dim
        if val[dim, :] <= split:
            tree = tree.left_leaf
        else:
            tree = tree.right_leaf
        return tree(val)


class Tree:
```

```python
    def __init__(self):
        self.split = None
        self.node = None
        self.left = None
        self.right = None
        self.left_leaf = None
        self.right_leaf = None
        self.number_mis_class_left = 0
        self.number_mis_class_right = 0
        # self.all_miss = 0


def plot(y, depth):
    x = range(1, depth + 1)
    plt.plot(x, y, marker='o')
    plt.xticks(x, rotation=0)
    plt.xlabel("Depth of Decision Tree")
    plt.ylabel("Num of Mis-classification")
    for a, b in zip(x, y):
        plt.text(a, b, '%.0f' % b, fontsize=11, ha='left', va='bottom')
    plt.title("Mis-classification VS Depth")
    plt.show()

if __name__ == "__main__":
    depth = 7
    mis_history = []
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/new_circle_data.csv'
    for d in range(1, depth + 1):
        decision_tree = Decision_Tree(file_path, d)
        mis_history.append(sum(decision_tree.misclassification))
    plot(mis_history, depth)
```

14.5 **Code up a multi-class classification tree**
Repeat the second experiment described in Example 14.4 by coding up a recursively defined multi-class classification tree. You need not reproduce Figure 14.12. Instead, measure and plot the number of misclassifications at each depth of your tree.

Sol: The figure below shows the trend of the number of misclassification in the decision tree as the depth increases.

In general, with the increase of the depth of the decision tree, the classification ability of the model will increase, but when the depth reaches a specific threshold, the over-fitting will occur.



The black numbers in the line chart represent the number of misclassification.

- **14-5 Code up a multi-class classification tree**

  - **Part 1 Basic structure of tree**

```
from autograd import numpy as np
import copy

left_his = []
right_his = []


class Stump:
   def __init__(self, x, y):
      self.x = x
      self.y = y
      self.make_stump()

   def counter(self, step, x, y):
      y_hat = step(x)[np.newaxis, :]
      vals, counts = np.unique(y, return_counts=True)
      balanced = 0
      for i in range(len(vals)):
         v = vals[i]
         c = counts[i]
         ind = np.argwhere(y == v)
         miss_val = 1
         if ind.size > 0:
            ind = [a[1] for a in ind]
            miss = np.argwhere(y_hat[:, ind] != y[:, ind])
            if miss.size > 0:
               miss = len([a[1] for a in miss])
               miss_val = (1 - miss / c)
         balanced += miss_val
      balanced = balanced / len(vals)
      return balanced

   def make_stump(self):
      N = np.shape(self.x)[0]
      P = np.size(self.y)
      acc_matrix_right = [0] * N
      acc_matrix_left = [0] * N
      best_split = np.inf
      best_dim = np.inf
```

```
best_val = -np.inf
best_left_leaf = []
best_right_leaf = []
best_left_ave = []
best_right_ave = []
best_step = []
c_vals, c_counts = np.unique(self.y, return_counts=True)
self.c_counts = c_counts

for n in range(N):
    x_n = copy.deepcopy(self.x[n, :])
    y_n = copy.deepcopy(self.y)

    sorted_inds = np.argsort(x_n, axis=0)
    x_n = x_n[sorted_inds]
    y_n = y_n[:, sorted_inds]
    for p in range(P - 1):
        if y_n[:, p] != y_n[:, p + 1] and x_n[p] != x_n[p + 1]:
            # compute split point
            split = (x_n[p] + x_n[p + 1]) / float(2)
            y_n_left = y_n[:, :p + 1]
            y_n_right = y_n[:, p + 1:]
            c_left_vals, c_left_counts = np.unique(y_n_left, return_counts=True)
            c_right_vals, c_right_counts = np.unique(y_n_right, return_counts=True)

            prop_left = []
            prop_right = []
            for i in range(np.size(c_vals)):
                val = c_vals[i]
                count = c_counts[i]

                val_ind = np.argwhere(c_left_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_left_counts[val_ind][0][0]
                prop_left.append(val_count / count)

                val_ind = np.argwhere(c_right_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_right_counts[val_ind][0][0]
                prop_right.append(val_count / count)

            prop_left = np.array(prop_left)
```

```
            best_left = np.argmax(prop_left)
            left_ave = c_vals[best_left]
            best_acc_left = prop_left[best_left]

            prop_right = np.array(prop_right)
            best_right = np.argmax(prop_right)
            right_ave = c_vals[best_right]
            best_acc_right = prop_right[best_right]
            # right = y_n_right.size / y_n.size
            val = (best_acc_left + best_acc_right) / 2

            left_leaf = lambda x, left_ave=left_ave, dim=n: np.array([left_ave for v in x[dim, :]])
            right_leaf = lambda x, right_ave=right_ave, dim=n: np.array([right_ave for v in
x[dim, :]])

            step = lambda x, split=split, left_ave=left_ave, right_ave=right_ave, dim=n: np.array(
                [(left_ave if v <= split else right_ave) for v in x[dim, :]])

            if val > best_val:
                acc_matrix_right = prop_right
                acc_matrix_left = prop_left
                best_left_leaf = copy.deepcopy(left_leaf)
                best_right_leaf = copy.deepcopy(right_leaf)
                best_dim = copy.deepcopy(n)
                best_split = copy.deepcopy(split)
                best_val = copy.deepcopy(val)
                best_step = copy.deepcopy(step)

        self.step = best_step
        self.left_leaf = best_left_leaf
        self.right_leaf = best_right_leaf
        self.dim = best_dim
        self.split = best_split

        # sort x_n and y_n according to ascending order in x_n
        sorted_inds = np.argsort(self.x[best_dim, :], axis=0)
        self.x = self.x[:, sorted_inds]
        self.y = self.y[:, sorted_inds]

        left_inds = np.argwhere(self.x[best_dim, :] <= best_split).flatten()
        right_inds = np.argwhere(self.x[best_dim, :] > best_split).flatten()

        self.left_x = self.x[:, left_inds]
        self.right_x = self.x[:, right_inds]
```

```
        self.left_y = self.y[:, left_inds]
        self.right_y = self.y[:, right_inds]
        self.number_mis_class_left = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[0]
        self.number_mis_class_right = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[1]
        right_his.append(self.number_mis_class_right)
        left_his.append(self.number_mis_class_left)

    def caculate_mis_class(self, prop_right, prop_left):
        leaf_label_ind_left = np.argmax(prop_left)
        left_label_count = self.c_counts[leaf_label_ind_left]
        leaf_label_ind_right = np.argmax(prop_right)
        right_label_count = self.c_counts[leaf_label_ind_right]
        mis_class_left = self.left_x.shape[1] - round(left_label_count * prop_left[leaf_label_ind_left])
        mis_class_right = self.right_x.shape[1] - round(right_label_count *
prop_right[leaf_label_ind_right])
        return mis_class_left, mis_class_right
```

➢ Part 2 Build Tree

```
from matplotlib import pyplot as plt
from mlrefined_libraries.nonlinear_superlearn_library.recursive_tree_lib.ClassificationTree import
ClassificationStump
import copy
import autograd.numpy as np

depth_count = 1


class Decision_Tree(object):
    def __init__(self, file_path, depth):
        data = np.loadtxt(file_path, delimiter=',')
        self.misclassification = []
        x = data[:-1, :]
        y = data[-1:, :]
        self.depth = depth
        self.tree = Tree()
        stump = ClassificationStump.Stump(x, y)
        self.build_tree(stump, self.tree, depth)

    def build_subtree(self, stump):
        best_split = stump.split
        best_dim = stump.dim
```

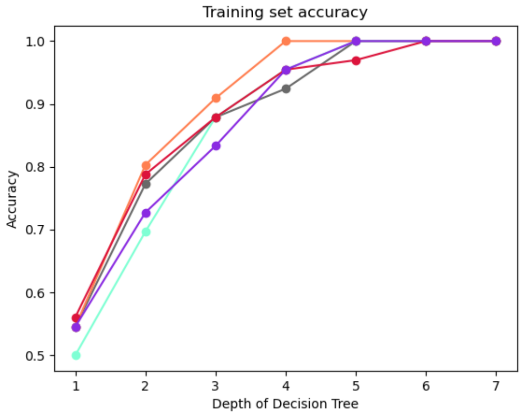
```
        left_x = stump.left_x
        right_x = stump.right_x
        left_y = stump.left_y
        right_y = stump.right_y
        left_stump = stump
        right_stump = stump
        if np.size(np.unique(left_y)) > 1:
            left_stump = ClassificationStump.Stump(left_x, left_y)
        else:
            right_stump.right_y = right_stump.left_y
            left_stump.number_mis_class_left = 0
            left_stump.number_mis_class_right = 0

        if np.size(np.unique(right_y)) > 1:
            right_stump = ClassificationStump.Stump(right_x, right_y)
        else:
            right_stump.left_y = right_stump.right_y
            right_stump.number_mis_class_left = 0
            right_stump.number_mis_class_right = 0

        return left_stump, right_stump

    def build_tree(self, stump, node, depth):
        if depth > 1:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
            node.right_leaf = stump.right_leaf
            node.step = stump.step
            node.number_mis_class_left = stump.number_mis_class_left
            node.number_mis_class_right = stump.number_mis_class_right
            left_stump, right_stump = self.build_subtree(stump)
            depth -= 1
            if left_stump.number_mis_class_right + left_stump.number_mis_class_left == 0 \
                    and right_stump.number_mis_class_right + right_stump.number_mis_class_left == 0:
                depth = 1
            node.left = Tree()
            node.right = Tree()
            return self.build_tree(right_stump, node.right, depth), self.build_tree(left_stump,
node.left, depth)
        else:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
```

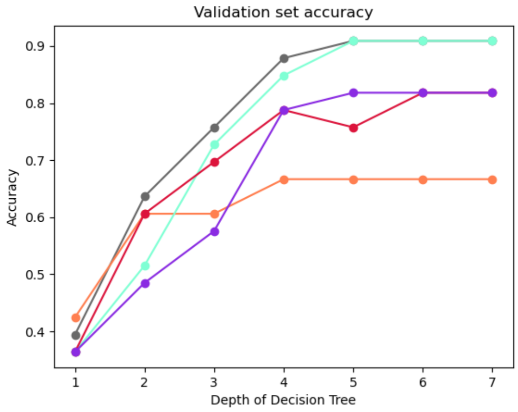
```
            node.right_leaf = stump.right_leaf
            node.step = stump.step
            node.number_mis_class_left = stump.number_mis_class_left
            node.number_mis_class_right = stump.number_mis_class_right
            self.misclassification.append(node.number_mis_class_left)
            self.misclassification.append(node.number_mis_class_right)
            print("miss_c is:" + str(self.misclassification))

    def evaluate_tree(self, val, depth):
        if depth > self.depth:
            return 'desired depth greater than depth of tree'
        tree = copy.deepcopy(self.tree)
        d = 0
        while d < depth:
            split = tree.split
            dim = tree.dim
            if val[dim, :] <= split:
                tree = tree.left
            else:
                tree = tree.right
            d += 1

        # get final leaf value
        split = tree.split
        dim = tree.dim
        if val[dim, :] <= split:
            tree = tree.left_leaf
        else:
            tree = tree.right_leaf
        return tree(val)


class Tree:
    def __init__(self):
        self.split = None
        self.node = None
        self.left = None
        self.right = None
        self.left_leaf = None
        self.right_leaf = None
        self.number_mis_class_left = 0
        self.number_mis_class_right = 0
        self.all_miss = 0
```

```python
def plot(y, depth):
    x = range(1, depth + 1)
    plt.plot(x, y, marker='o')
    plt.xticks(x, rotation=0)
    plt.xlabel("Depth of Decision Tree")
    plt.ylabel("Num of Mis-classification")
    for a, b in zip(x, y):
        plt.text(a, b, '%.0f' % b, fontsize=11, ha='left', va='bottom')
    plt.title("Mis-classification VS Depth")
    plt.show()


if __name__ == "__main__":
    depth = 7
    mis_history = []
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/3_layercake_data.csv'
    for d in range(1, depth + 1):
        decision_tree = Decision_Tree(file_path, d)
        mis_history.append(sum(decision_tree.misclassification))
    plot(mis_history, depth)
```

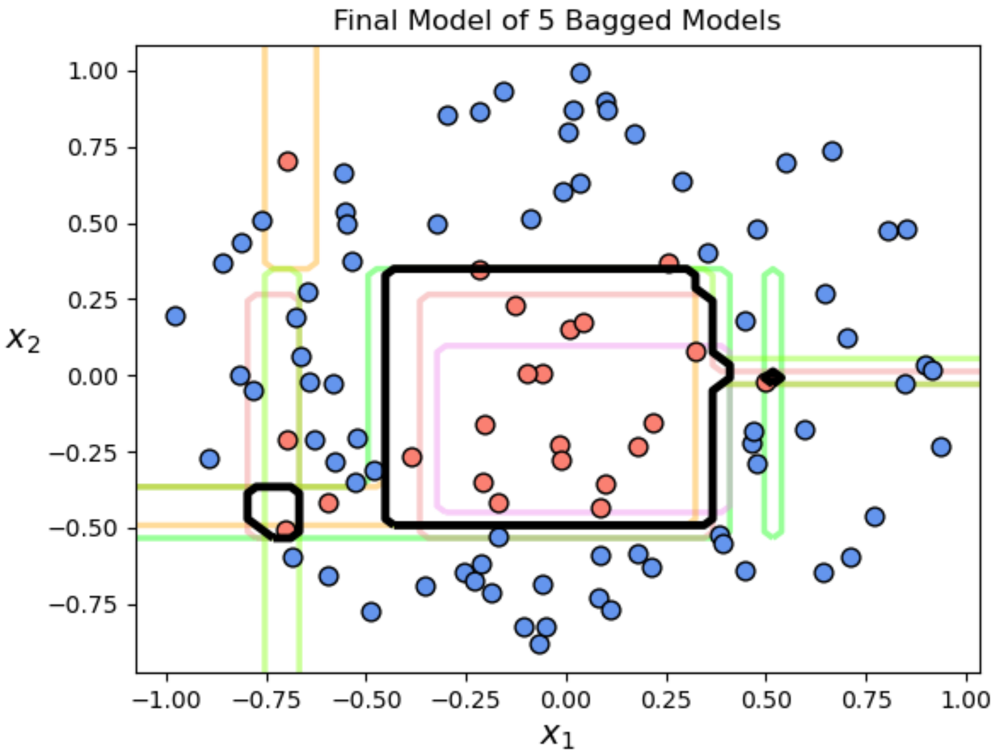
14.8 **Random forests**
Repeat the experiment described in Example 14.7 by coding up a random forest built from classification trees. You need not reproduce Figure 14.15. However, you can verify that your implementation is working properly by checking that

Sol: As can be seen from the figure below, the accuracy of training set is significantly higher than that of validation set, which indicate that with the increase of depth of decision tree, most of the individual trees overfit the data.





(mode of the five individual trees)
The ensembled model, as illustrated below, is not overfitting



- ## 14-5 Random Forests

  - ### Part 1 Basic structure of tree

```
from autograd import numpy as np
import copy

left_his = []
right_his = []


class Stump:
   def __init__(self, x, y):
      self.x = x
      self.y = y
      self.make_stump()

   def counter(self, step, x, y):
      y_hat = step(x)[np.newaxis, :]
      vals, counts = np.unique(y, return_counts=True)
      balanced = 0
      for i in range(len(vals)):
         v = vals[i]
         c = counts[i]
         ind = np.argwhere(y == v)
         miss_val = 1
         if ind.size > 0:
            ind = [a[1] for a in ind]
            miss = np.argwhere(y_hat[:, ind] != y[:, ind])
            if miss.size > 0:
               miss = len([a[1] for a in miss])
               miss_val = (1 - miss / c)
         balanced += miss_val
      balanced = balanced / len(vals)
      return balanced

   def make_stump(self):
      N = np.shape(self.x)[0]
      P = np.size(self.y)
      acc_matrix_right = [0] * N
      acc_matrix_left = [0] * N
      best_split = np.inf
      best_dim = np.inf
```

```
best_val = -np.inf
best_left_leaf = []
best_right_leaf = []
best_left_ave = []
best_right_ave = []
best_step = []
c_vals, c_counts = np.unique(self.y, return_counts=True)
self.c_counts = c_counts

for n in range(N):
    x_n = copy.deepcopy(self.x[n, :])
    y_n = copy.deepcopy(self.y)

    sorted_inds = np.argsort(x_n, axis=0)
    x_n = x_n[sorted_inds]
    y_n = y_n[:, sorted_inds]
    for p in range(P - 1):
        if y_n[:, p] != y_n[:, p + 1] and x_n[p] != x_n[p + 1]:
            # compute split point
            split = (x_n[p] + x_n[p + 1]) / float(2)
            y_n_left = y_n[:, :p + 1]
            y_n_right = y_n[:, p + 1:]
            c_left_vals, c_left_counts = np.unique(y_n_left, return_counts=True)
            c_right_vals, c_right_counts = np.unique(y_n_right, return_counts=True)

            prop_left = []
            prop_right = []
            for i in range(np.size(c_vals)):
                val = c_vals[i]
                count = c_counts[i]

                val_ind = np.argwhere(c_left_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_left_counts[val_ind][0][0]
                prop_left.append(val_count / count)

                # check right side
                val_ind = np.argwhere(c_right_vals == val)
                val_count = 0
                if np.size(val_ind) > 0:
                    val_count = c_right_counts[val_ind][0][0]
                prop_right.append(val_count / count)
```

```
            # array it
            prop_left = np.array(prop_left)
            best_left = np.argmax(prop_left)
            left_ave = c_vals[best_left]
            best_acc_left = prop_left[best_left]
            prop_right = np.array(prop_right)
            best_right = np.argmax(prop_right)
            right_ave = c_vals[best_right]
            best_acc_right = prop_right[best_right]
            # right = y_n_right.size / y_n.size
            val = (best_acc_left + best_acc_right) / 2

            left_leaf = lambda x, left_ave=left_ave, dim=n: np.array([left_ave for v in x[dim, :]])
            right_leaf = lambda x, right_ave=right_ave, dim=n: np.array([right_ave for v in
x[dim, :]])

            step = lambda x, split=split, left_ave=left_ave, right_ave=right_ave, dim=n: np.array(
                [(left_ave if v <= split else right_ave) for v in x[dim, :]])

            if val > best_val:
                acc_matrix_right = prop_right
                acc_matrix_left = prop_left
                best_left_leaf = copy.deepcopy(left_leaf)
                best_right_leaf = copy.deepcopy(right_leaf)

                best_dim = copy.deepcopy(n)
                best_split = copy.deepcopy(split)
                best_val = copy.deepcopy(val)
                best_left_ave = copy.deepcopy(left_ave)
                best_right_ave = copy.deepcopy(right_ave)
                best_step = copy.deepcopy(step)

        self.step = best_step
        self.left_leaf = best_left_leaf
        self.right_leaf = best_right_leaf
        self.dim = best_dim
        self.split = best_split

        sorted_inds = np.argsort(self.x[best_dim, :], axis=0)
        self.x = self.x[:, sorted_inds]
        self.y = self.y[:, sorted_inds]

        left_inds = np.argwhere(self.x[best_dim, :] <= best_split).flatten()
        right_inds = np.argwhere(self.x[best_dim, :] > best_split).flatten()
```

```python
        self.left_x = self.x[:, left_inds]
        self.right_x = self.x[:, right_inds]
        self.left_y = self.y[:, left_inds]
        self.right_y = self.y[:, right_inds]
        self.number_mis_class_left = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[0]
        self.number_mis_class_right = self.caculate_mis_class(acc_matrix_right, acc_matrix_left)[1]
        right_his.append(self.number_mis_class_right)
        left_his.append(self.number_mis_class_left)

    def caculate_mis_class(self, prop_right, prop_left):
        leaf_label_ind_left = np.argmax(prop_left)
        left_label_count = self.c_counts[leaf_label_ind_left]
        leaf_label_ind_right = np.argmax(prop_right)
        right_label_count = self.c_counts[leaf_label_ind_right]
        mis_class_left = self.left_x.shape[1] - round(left_label_count * prop_left[leaf_label_ind_left])
        mis_class_right = self.right_x.shape[1] - round(right_label_count *
prop_right[leaf_label_ind_right])
        return mis_class_left, mis_class_right
```

➢ Part 2 Build Tree

```python
from matplotlib import pyplot as plt
from mlrefined_libraries.nonlinear_superlearn_library.recursive_tree_lib.ClassificationTree import
ClassificationStump
import autograd.numpy as np
import copy


class Tree:
    def __init__(self):
        self.split = None
        self.node = None
        self.left = None
        self.right = None
        self.left_leaf = None
        self.right_leaf = None
        self.number_mis_class_left = 0
        self.number_mis_class_right = 0
        self.all_miss = 0


class Random_Forest_Algorithm:
```

```python
def __init__(self, csvname, depth, train_portion):
    data = np.loadtxt(csvname, delimiter=',')
    self.x = data[:-1, :]
    self.y = data[-1:, :]
    self.depth = depth
    self.colors = ['salmon', 'cornflowerblue', 'lime', 'bisque', 'mediumaquamarine', 'b', 'm', 'g']
    self.plot_colors = ['lime', 'violet', 'orange', 'lightcoral', 'chartreuse', 'aqua', 'deeppink']

    self.make_train_val_split(train_portion)

    # build root regression stump
    self.tree = Tree()
    stump = ClassificationStump.Stump(self.x_train, self.y_train)

    # build remainder of tree
    self.build_tree(stump, self.tree, depth)

    # compute train / valid errors
    self.compute_train_val_accuracies()
    self.best_depth = np.argmax(self.valid_accuracies)

def make_train_val_split(self, train_portion):
    self.train_portion = train_portion
    r = np.random.permutation(self.x.shape[1])
    train_num = int(np.round(train_portion * len(r)))
    self.train_inds = r[:train_num]
    self.valid_inds = r[train_num:]
    self.x_train = self.x[:, self.train_inds]
    self.x_valid = self.x[:, self.valid_inds]
    self.y_train = self.y[:, self.train_inds]
    self.y_valid = self.y[:, self.valid_inds]

def build_subtree(self, stump):
    # get params from input stump
    best_split = stump.split
    best_dim = stump.dim
    left_x = stump.left_x
    right_x = stump.right_x
    left_y = stump.left_y
    right_y = stump.right_y

    left_stump = stump
    right_stump = stump
```

```
        if np.size(np.unique(left_y)) > 1:
            left_stump = ClassificationStump.Stump(left_x, left_y)
        if np.size(np.unique(right_y)) > 1:
            right_stump = ClassificationStump.Stump(right_x, right_y)
        return left_stump, right_stump

    def build_tree(self, stump, node, depth):
        if depth > 1:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
            node.right_leaf = stump.right_leaf
            node.step = stump.step
            left_stump, right_stump = self.build_subtree(stump)

            node.left = Tree()
            node.right = Tree()
            depth -= 1
            return self.build_tree(left_stump, node.left, depth), self.build_tree(right_stump,
node.right, depth)
        else:
            node.split = stump.split
            node.dim = stump.dim
            node.left_leaf = stump.left_leaf
            node.right_leaf = stump.right_leaf
            node.step = stump.step

    def compute_train_val_accuracies(self):
        self.train_accuracies = []
        self.valid_accuracies = []
        for j in range(self.depth):
            # compute training error
            train_evals = np.array([self.predict(v[:, np.newaxis], depth=j) for v in self.x_train.T]).T
            valid_evals = np.array([self.predict(v[:, np.newaxis], depth=j) for v in self.x_valid.T]).T

            # compute cost
            train_miss = 0
            if self.y_train.size > 0:
                train_miss = 1 - len(np.argwhere(train_evals != self.y_train)) / self.y_train.size
            valid_miss = 0
            if self.y_valid.size > 0:
                valid_miss = 1 - len(np.argwhere(valid_evals != self.y_valid)) / self.y_valid.size

            self.train_accuracies.append(train_miss)
```

```
        self.valid_accuracies.append(valid_miss)

    def predict(self, val, **kwargs):
        depth = self.depth
        if 'depth' in kwargs:
            depth = kwargs['depth']

        # search tree
        tree = copy.deepcopy(self.tree)
        d = 0
        while d < depth:
            split = tree.split
            dim = tree.dim
            if val[dim, :] <= split:
                tree = tree.left
            else:
                tree = tree.right
            d += 1

        # get final leaf value
        split = tree.split
        dim = tree.dim
        if val[dim, :] <= split:
            tree = tree.left_leaf
        else:
            tree = tree.right_leaf

        # return evaluation
        return tree(val)

    def evaluate_tree(self, val, depth):
        if depth > self.depth:
            return ('desired depth greater than depth of tree')
        tree = copy.deepcopy(self.tree)
        d = 0
        while d < depth:
            split = tree.split
            dim = tree.dim
            if val[dim, :] <= split:
                tree = tree.left
            else:
                tree = tree.right
            d += 1
```

```python
        # get final leaf value
        split = tree.split
        dim = tree.dim
        if val[dim, :] <= split:
            tree = tree.left_leaf
        else:
            tree = tree.right_leaf
        return tree(val)

    def draw_fused_model(self, runs):
        # get visual boundary
        xmin1 = np.min(self.x[0, :])
        xmax1 = np.max(self.x[0, :])
        xgap1 = (xmax1 - xmin1) * 0.05
        xmin1 -= xgap1
        xmax1 += xgap1
        xmin2 = np.min(self.x[1, :])
        xmax2 = np.max(self.x[1, :])
        xgap2 = (xmax2 - xmin2) * 0.05
        xmin2 -= xgap2
        xmax2 += xgap2
        ind0 = np.argwhere(self.y == +1)
        ind0 = [v[1] for v in ind0]
        plt.scatter(self.x[0, ind0], self.x[1, ind0], s=60, color=self.colors[0], edgecolor='k',
linewidth=1, zorder=3)
        ind1 = np.argwhere(self.y == -1)
        ind1 = [v[1] for v in ind1]
        plt.scatter(self.x[0, ind1], self.x[1, ind1], s=60, color=self.colors[1], edgecolor='k',
linewidth=1, zorder=3)
        plt.xlim([xmin1, xmax1])
        plt.ylim([xmin2, xmax2])
        plt.title("Final Model of " + str(num_trees) + " Bagged Models")
        plt.xlabel(r'$x_1$', fontsize=14)
        plt.ylabel(r'$x_2$', rotation=0, fontsize=14, labelpad=10)
        s1 = np.linspace(xmin1, xmax1, 50)
        s2 = np.linspace(xmin2, xmax2, 50)
        a, b = np.meshgrid(s1, s2)
        a = np.reshape(a, (np.size(a), 1))
        b = np.reshape(b, (np.size(b), 1))
        h = np.concatenate((a, b), axis=1)
        a.shape = (np.size(s1), np.size(s2))
        b.shape = (np.size(s1), np.size(s2))
        t_ave = []
        for k in range(len(runs)):
```

```
            tree = runs[k]
            depth = tree.best_depth
            t = []
            for val in h:
                val = val[:, np.newaxis]
                out = tree.evaluate_tree(val, depth)
                t.append(out)
            t = np.array(t)
            t.shape = (np.size(s1), np.size(s2))
            col = np.random.rand(1, 3)
            plt.contour(s1, s2, t, linewidths=2.5, levels=[0], colors=self.plot_colors[k], zorder=2,
alpha=0.4)
            t_ave.append(t)
        t_ave = np.array(t_ave)
        t_ave1 = np.median(t_ave, axis=0)
        plt.contour(s1, s2, t_ave1, linewidths=3.5, levels=[0], colors='k', zorder=4, alpha=1)
        plt.show()


def plot(y, label):
    x = range(1, len(y[1]) + 1)
    colors = ['dimgray', 'coral', 'aquamarine', 'crimson', 'blueviolet', 'chartreuse']
    plt.title(label)
    for i in range(len(y)):
        plt.plot(x, y[i], marker='o', color=colors[i])
    plt.xticks(x, rotation=0)
    plt.xlabel("Depth of Decision Tree")
    plt.ylabel("Accuracy")
    plt.show()


if __name__ == "__main__":
    file_path = '../mlrefined_datasets/nonlinear_superlearn_datasets/new_circle_data.csv'
    trees = []
    train_acc = []
    valid_acc = []
    num_trees = 5
    depth = 7
    train_portion = 0.66
    for i in range(num_trees):
        print("training fold: " + str(i))
        tree = Random_Forest_Algorithm(file_path, depth, train_portion=train_portion)
        trees.append(tree)
        train_acc.append(tree.train_accuracies)
```

```
    valid_acc.append(tree.valid_accuracies)
# Compare the acc of training_set and validation_set
plot(train_acc, label='Training set accuracy')
plot(valid_acc, label='Validation set accuracy')
# Draw 5+1 models all in one
tree = Random_Forest_Algorithm(file_path, depth, train_portion=1)
tree.draw_fused_model(runs=trees)
```