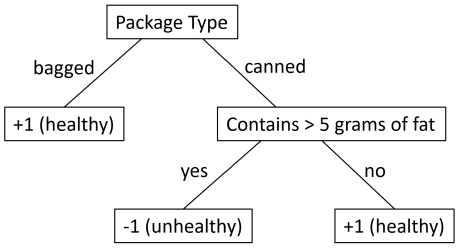
Machine Learning & Data Mining
Caltech CS/CNS/EE 155
Homework 3

Changhao Xu
UID: 2103530
January 30th, 2020

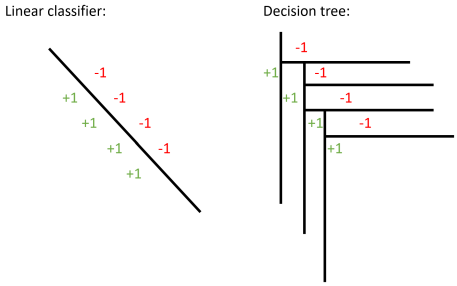
---

1 Decision Trees

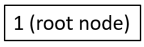
Question A: At root node, $p_{s'}$ = 0.75 and S' = 4, entropy = 2.249.
At 1st layer:
Package type: bagged = $\{+1, +1\}$, canned = $\{+1, -1\}$, entropy = 1.386
Unit price $>$ \$5: yes = $\{+1, -1\}$, no = $\{+1, +1\}$, entropy = 1.386
Contains $>$ 5 grams of fat: yes = $\{+1, -1\}$, no = $\{+1, +1\}$, entropy = 1.386
Here, since entropy is same, we choose package type as 1st layer.
At 2nd layer:
Unit price $>$ \$5: yes = -1, no = +1, entropy = 0
Contains $>$ 5 grams of fat: yes = -1, no = +1,entropy = 0
Here, since entropy is same, we choose Contains $>$ 5 grams of fat as 2nd layer.



Question B: Decision tree is not always preferred for classification problems compared to linear classifier. For example:

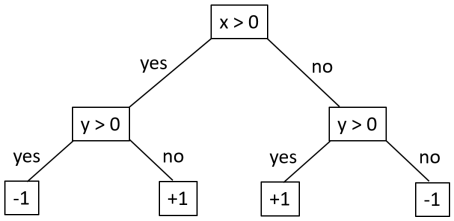


Question C: i. At root node, $p_{s'}$ = 0.5, Impurity via Gini index = $4 \times (1 - 0.5^2 - 0.5^2) = 2$.



Suppose split on X/Y axis, then impurity becomes $2 \times (1 - 0.5^2 - 0.5^2) + 2 \times (1 - 0.5^2 - 0.5^2) = 2$. So further split won't reduce impurity. Therefore, decision tree is just one root node, and classification error = 0.5.

ii.



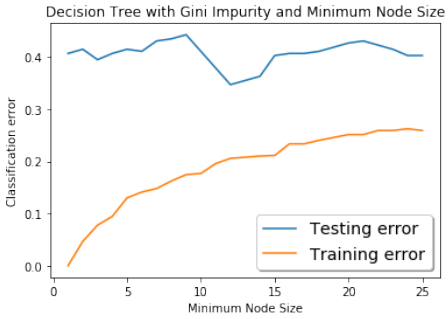
Impurity measure $L(S') = |S'|^2 \times \left[1 - p_{s'}^2, - (1 - p_{s'})^2\right]$. This will lead to an initial error of 8 at root node, then error becomes 4 at 1st layer, and becomes 0 at leaf node. The pros of this error measure is that it can successfully split a data set that have same percentage of correct and incorrect classifications before and after the split. The cons of this error measure is that it highly depends on the number of classified points and misclassified points, and may lead to overfitting due to encouragement of splitting.

iii. The largest number of internal nodes will be 99 to achieve zero classification training error. Suppose 100 points in 1D, which are labeled as 1, -1, 1, -1, ... Then it needs each data point to be in a unique split. Therefore, the decision tree will have (100-1) = 99 nodes.

Question D: As described in C(iii), N data points will need (N-1) internal nodes at most. Then worst-case complexity = O(N*D).

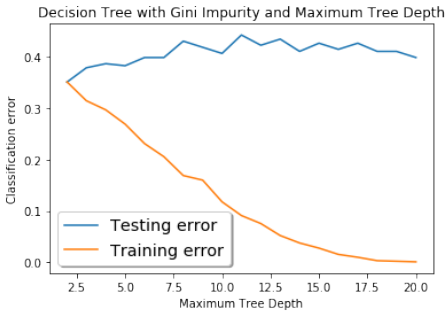
2 Overfitting Decision Trees

Question A:



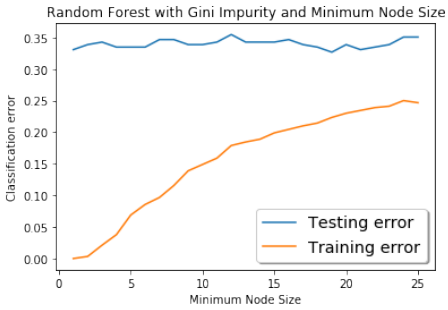
Changhao Xu
UID: 2103530

Question B:

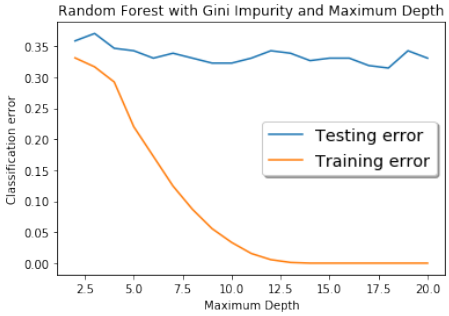


Question C: For the minimal leaf node size, 12 has minimized test error. For maximum depth parameters, 2 has minimized test error. Early stopping will improve the performance of a decision tree model. In Question A, when minimal leaf node size = 1 (no early stopping), test error is high. Similarly, in Question B, when maximum depth = 20 (minimized early stopping), test error is high.

Question D:



Question E:



Question F: For the minimal leaf node size, 19 minimizes the random forest test error. For maximum depth parameters, 18 minimizes the random forest test error. Early stopping will not improve, or even impair the performance of a random forest model. In Question D, when minimal leaf node size = 1 (no early stopping), test error is stable and early stopping doesn't have much effect on lowering test error. Similarly, in Question E, when maximum depth = 20 (minimized early stopping), test error is low and early stopping increases test error.

Question G: Test error for random forest is lower and the curve is smoother than that of decision tree model. Decision tree is very likely to overfitting, so early stopping is desired; while random forest samples both data and features, and thus reduces variance, so early stopping doesn't have much effect.

3 The AdaBoost Algorithm

Question A:
If y_i is correctly classified, i.e. $-y_i f(x_i) < 0$. Then $exp(-y_i f(x_i)) > 0$, while $\mathbb{1}(H(x_i) \neq y_i) = 0$.
If y_i is incorrectly classified, i.e. $-y_i f(x_i) > 0$. Then $exp(-y_i f(x_i)) > 1$, while $\mathbb{1}(H(x_i) \neq y_i) = 1$.
Therefore, $exp(-y_i f(x_i)) \geq \mathbb{1}(H(x_i) \neq y_i)$ for $\forall i \in N$. And thus

$$E = \frac{1}{N}\sum_{i=1}^{N} \exp(-y_i f(x_i)) \geq \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(H(x_i) \neq y_i)$$

Question B: In lecture 6,

$$D_{t+1}(i) = \frac{D_t(i) exp\left\{-\alpha_t y_i h_t(x_i)\right\}}{Z_t}$$

, so

$$D_{T+1}(i) = D_1(i) \prod_{t=1}^{T} \frac{exp\left\{-\alpha_t y_i h_t(x_i)\right\}}{Z_t}$$

, where $D_1(i) = \frac{1}{N}$. Therefore,

$$D_{T+1}(i) = \frac{1}{N} \prod_{t=1}^{T} \frac{exp\left\{-\alpha_t y_i h_t(x_i)\right\}}{Z_t}$$

, where Z_t is the normalization factor

$$Z_t = \sum_{i=1}^{N} D_t(i) exp\left\{-\alpha_t y_i h_t(x_i)\right\}$$

Question C:

$$E = \frac{1}{N}\sum_{i=1}^{N} \exp(-y_i f(x_i))$$

, where

$$f(x_i) = \sum_{t=1}^{T} \alpha_t h_t(x_i)$$

, so

$$E = \frac{1}{N}\sum_{i=1}^{N}\exp(-y_i\sum_{t=1}^{T}\alpha_t h_t(x_i))$$

Question D: From Question B,

$$D_{T+1}(i) = \frac{1}{N}\prod_{t=1}^{T}\frac{exp\left\{-\alpha_t y_i h_t(x_i)\right\}}{Z_t} = \frac{1}{N}\frac{\prod_{t=1}^{T} exp\left\{-\alpha_t y_i h_t(x_i)\right\}}{\prod_{t=1}^{T} Z_t} = \frac{1}{N}\frac{exp\left\{\sum_{t=1}^{T} -\alpha_t y_i h_t(x_i)\right\}}{\prod_{t=1}^{T} Z_t}$$

Therefore,

$$\begin{aligned} E &= \frac{1}{N}\sum_{i=1}^{N}\exp(-y_i\sum_{t=1}^{T}\alpha_t h_t(x_i)) = \sum_{i=1}^{N}\frac{1}{N}\exp(\sum_{t=1}^{T}-\alpha_t y_i h_t(x_i)) \\ &= \sum_{i=1}^{N} D_{T+1}(i)\prod_{t=1}^{T} Z_t \end{aligned}$$

Since $\sum_{i=1}^{N} D_t(i) = 1$,

$$E = \prod_{t=1}^{T} Z_t$$

.

Question E:

$$Z_t = \sum_{i=1}^{N} D_t(i) exp\left\{-\alpha_t y_i h_t(x_i)\right\}$$

If y_i is correctly classified, i.e. $-y_i h_t(x_i) = -1$. $Z_t = \sum_{i=1}^{N} D_t(i) exp(-\alpha_t) = exp(-\alpha_t)$.
In this case, $\epsilon_t = \sum_{i=1}^{N} D_t(i)\mathbb{1}(H(x_i) \neq y_i) = 0$, so $Z_t = (1-\epsilon_t)exp(-\alpha_t) + \epsilon_t exp(\alpha_t)$
If y_i is incorrectly classified, i.e. $-y_i h_t(x_i) = 1$. $Z_t = \sum_{i=1}^{N} D_t(i) exp(\alpha_t) = exp(\alpha_t)$.
In this case, $\epsilon_t = \sum_{i=1}^{N} D_t(i)\mathbb{1}(H(x_i) \neq y_i) = 1$, so $Z_t = (1-\epsilon_t)exp(-\alpha_t) + \epsilon_t exp(\alpha_t)$.

In both cases, $Z_t = (1-\epsilon_t)exp(-\alpha_t) + \epsilon_t exp(\alpha_t)$

Question F:

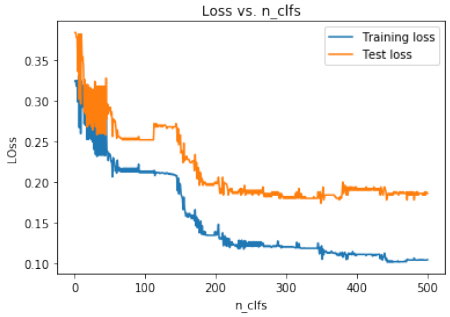
$$\begin{aligned} \frac{dZ_t}{d\alpha_t} &= -\alpha_t(1-\epsilon_t)exp(-\alpha_t) + \alpha_t\epsilon_t exp(\alpha_t) = 0 \\ \alpha_t\epsilon_t exp(\alpha_t) &= \alpha_t(1-\epsilon_t)exp(-\alpha_t) \\ \epsilon_t exp(\alpha_t) &= (1-\epsilon_t)exp(-\alpha_t) \\ \epsilon_t exp(2\alpha_t) &= (1-\epsilon_t) \\ \alpha_t &= \frac{1}{2}ln(\frac{1-\epsilon_t}{\epsilon_t}) \end{aligned}$$

Question G:

Changhao Xu
UID: 2103530
January 30th, 2020

See Jupyter notebook. Loss curve for Gradient Boosting:

Loss curve for Adaboost:



Question H : Gradient boosting has a smoother training and testing loss. Training loss of Gradient boosting approaches 0 steadily, while test loss only decreased initially and then slowly increase, which is due to overfitting. Adaboost has a rough training and testing loss, which is due to the Decision tree classifier used and 0/1 loss. Training loss of Adaboost approaches 0, and test loss also decreases until a steady state.

Question I: Final loss value for Gradient boosting is 0.264, and 0.186 for Adaboost. Adaboost performed better on the classification dataset.

Question J : Dataset weights are the largest at the decision boundary (where red and blue points overlap), and smallest when data point is furthest away from the decision boundary.

2_notebook

January 28, 2020

1 Problem 2

Import statements and function to load data:

[1]:
```
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import numpy as np

# Seed the random number generator:
np.random.seed(1)

def load_data(filename, skiprows = 1):
    """
    Function loads data stored in the file filename and returns it as a numpy␣
 ↪ndarray.

    Inputs:
        filename: given as a string.

    Outputs:
        Data contained in the file, returned as a numpy ndarray
    """
    return np.loadtxt(filename, skiprows=skiprows, delimiter=',')
```

Load the data and divide it into training and validation sets:

[2]:
```
# The number 24 in the next line corresponds to the number of header lines
X = load_data('data/messidor_features.arff', 24)

data = X[:, :-1]
diag = X[:, -1]

train_size = 900

train_data = data[0:train_size]
train_label = diag[0:train_size]
test_data = data[train_size:]
test_label = diag[train_size:]
```

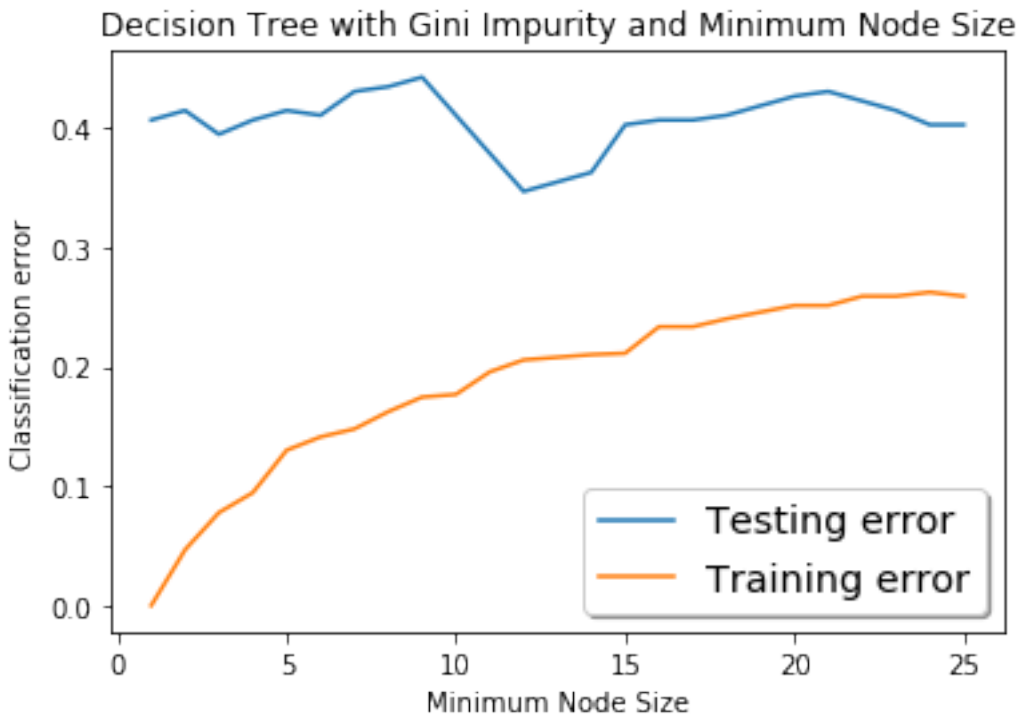
1.1 Problem 2A: Decision Trees with Minimum Leaf Size Stopping Criterion

Fill in the two functions below:

[3]:
```
def classification_err(y, real_y):
    """
    This function returns the classification error between two equally-sized␣
 ↪vectors of
    labels; this is the fraction of samples for which the labels differ.

    Inputs:
        y: (N, ) shaped array of predicted labels
        real_y: (N, ) shaped array of true labels
    Output:
        Scalar classification error
    """
    err = 0
    for i in range(real_y.shape[0]):
        if y[i] != real_y[i]:
            err += 1
    return float(err) / real_y.shape[0]

    pass

def eval_tree_based_model_min_samples(clf, min_samples_leaf, X_train, y_train,␣
 ↪X_test, y_test):
    """
    This function evaluates the given classifier (either a decision tree or␣
 ↪random forest) at all of the
    minimum leaf size parameters in the vector min_samples_leaf, using the␣
 ↪given training and testing
    data. It returns two vector, with the training and testing classification␣
 ↪errors.

    Inputs:
        clf: either a decision tree or random forest classifier object
        min_samples_leaf: a (T, ) vector of all the min_samples_leaf stopping␣
 ↪condition parameters
                          to test, where T is the number of parameters to␣
 ↪test
        X_train: (N, D) matrix of training samples.
        y_train: (N, ) vector of training labels.
        X_test: (N, D) matrix of test samples
        y_test: (N, ) vector of test labels
    Output:
        train_err: (T, ) vector of classification errors on the training data
        test_err: (T, ) vector of classification errors on the test data
    """
```

```
    train_err = np.array([])
    test_err = np.array([])

    for i in range(len(min_samples_leaf)):
        clf.set_params(min_samples_leaf = min_samples_leaf[i])
        clf.fit(X_train, y_train)

        err_1 = classification_err(clf.predict(X_train), y_train)
        train_err = np.append(train_err, err_1)

        err_2 = classification_err(clf.predict(X_test), y_test)
        test_err = np.append(test_err, err_2)

    return train_err, test_err

    pass
```

[4]:
```
# Seed the random number generator:
np.random.seed(1)

min_samples_leaf = np.arange(1, 26)
clf = tree.DecisionTreeClassifier(criterion='gini')

train_err, test_err = eval_tree_based_model_min_samples(clf, min_samples_leaf,␣
 ↪train_data,
                                                        train_label, test_data,␣
 ↪test_label)

plt.figure()
plt.plot(min_samples_leaf, test_err, label='Testing error')
plt.plot(min_samples_leaf, train_err, label='Training error')
plt.xlabel('Minimum Node Size')
plt.ylabel('Classification error')
plt.title('Decision Tree with Gini Impurity and Minimum Node Size')
plt.legend(loc=0, shadow=True, fontsize='x-large')
plt.show()

print('Test error minimized at min_samples_leaf = %i' % min_samples_leaf[np.
 ↪argmin(test_err)])
```




```
Test error minimized at min_samples_leaf = 12
```

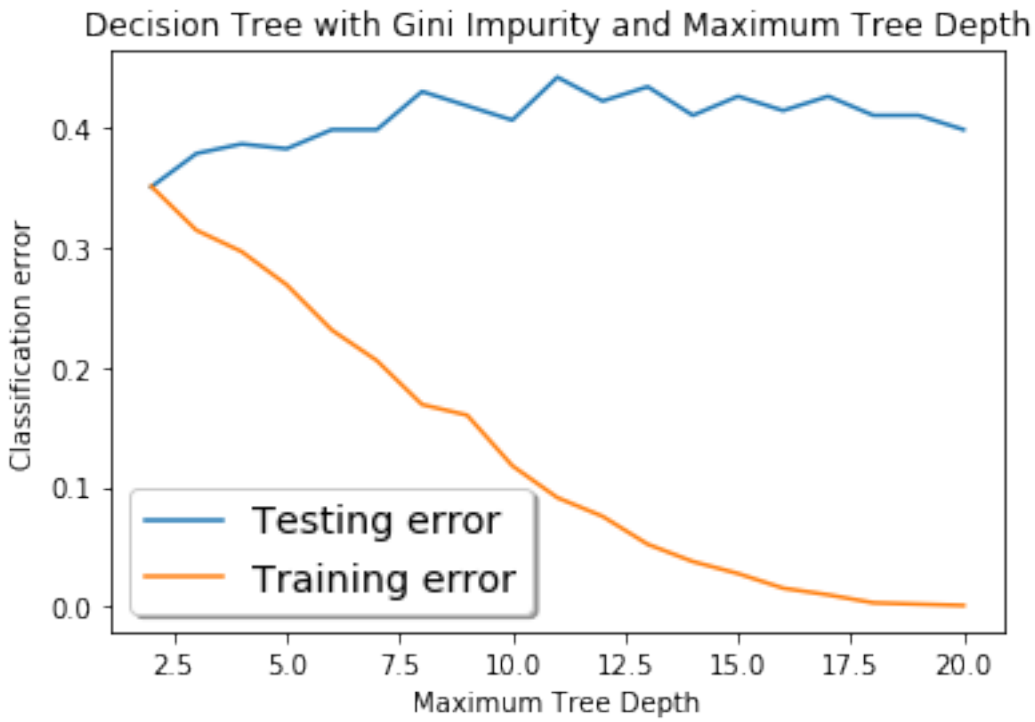
1.2 Problem 2B: Decision Trees with Maximum Depth Stopping Criterion

Fill in the function below:

[5]:
```
def eval_tree_based_model_max_depth(clf, max_depth, X_train, y_train, X_test,␣
↪y_test):
    """
    This function evaluates the given classifier (either a decision tree or␣
↪random forest) at all of the
    maximum tree depth parameters in the vector max_depth, using the given␣
↪training and testing
    data. It returns two vector, with the training and testing classification␣
↪errors.

    Inputs:
        clf: either a decision tree or random forest classifier object
        max_depth: a (T, ) vector of all the max_depth stopping condition␣
↪parameters
                            to test, where T is the number of parameters to␣
↪test
        X_train: (N, D) matrix of training samples.
        y_train: (N, ) vector of training labels.
```

```
        X_test: (N, D) matrix of test samples
        y_test: (N, ) vector of test labels
    Output:
        train_err: (T, ) vector of classification errors on the training data
        test_err: (T, ) vector of classification errors on the test data
    """

    train_err = np.array([])
    test_err = np.array([])

    for i in range(len(max_depth)):
        clf.set_params(max_depth = max_depth[i])
        clf.fit(X_train, y_train)

        err_1 = classification_err(clf.predict(X_train), y_train)
        train_err = np.append(train_err, err_1)

        err_2 = classification_err(clf.predict(X_test), y_test)
        test_err = np.append(test_err, err_2)

    return train_err, test_err

    pass
```

[6]:
```
# Seed the random number generator:
np.random.seed(1)

max_depth = np.arange(2, 21)

clf = tree.DecisionTreeClassifier(criterion='gini')

train_err, test_err = eval_tree_based_model_max_depth(clf, max_depth,␣
 ↪train_data,
                                                      train_label, test_data,␣
 ↪test_label)

plt.figure()
plt.plot(max_depth, test_err, label='Testing error')
plt.plot(max_depth, train_err, label='Training error')
plt.xlabel('Maximum Tree Depth')
plt.ylabel('Classification error')
plt.title('Decision Tree with Gini Impurity and Maximum Tree Depth')
plt.legend(loc=0, shadow=True, fontsize='x-large')
plt.show()

print('Test error minimized at max_depth = %i' % max_depth[np.argmin(test_err)])
```

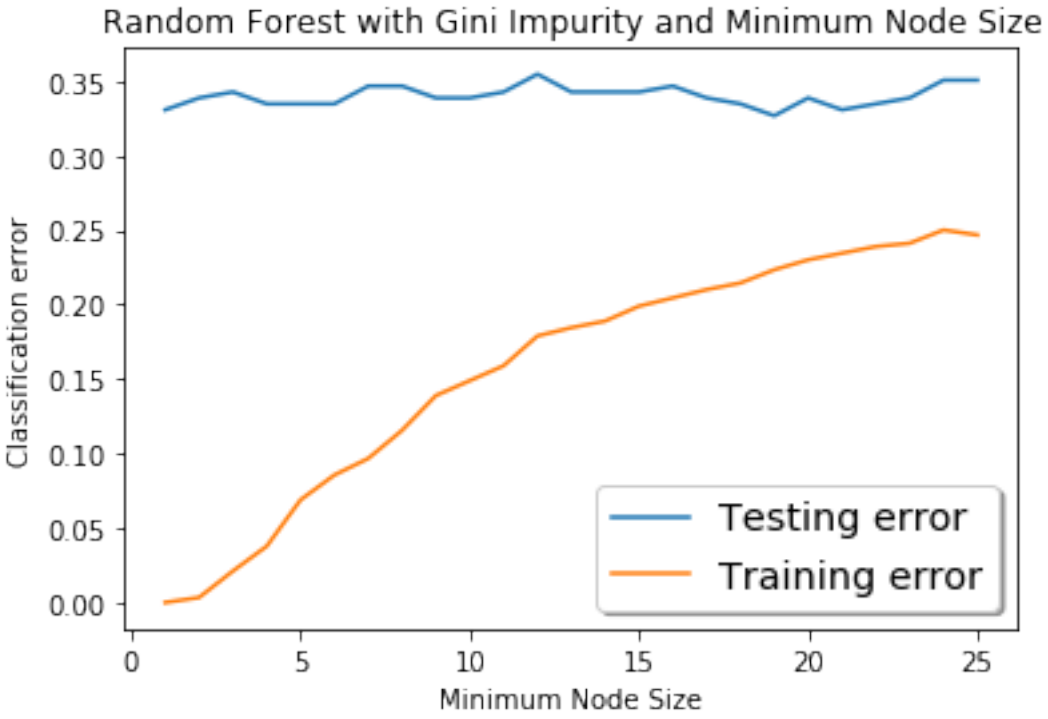



```
Test error minimized at max_depth = 2
```

1.3 Problem 2D: Random Forests with Minimum Leaf Size Stopping Criterion

[7]:
```
# Seed the random number generator:
np.random.seed(1)

n_estimators = 1000
clf = RandomForestClassifier(n_estimators = n_estimators, criterion = 'gini')

min_samples_leaf = np.arange(1, 26)

train_err, test_err = eval_tree_based_model_min_samples(clf, min_samples_leaf,␣
 ↪train_data,
                                                         train_label, test_data,␣
 ↪test_label)

plt.figure()
plt.plot(min_samples_leaf, test_err, label='Testing error')
plt.plot(min_samples_leaf, train_err, label='Training error')
plt.xlabel('Minimum Node Size')
plt.ylabel('Classification error')
plt.title('Random Forest with Gini Impurity and Minimum Node Size')
```

```
plt.legend(loc=0, shadow=True, fontsize='x-large')
plt.show()

print('Test error minimized at min_samples_leaf = %i' % min_samples_leaf[np.
 ↪argmin(test_err)])
```

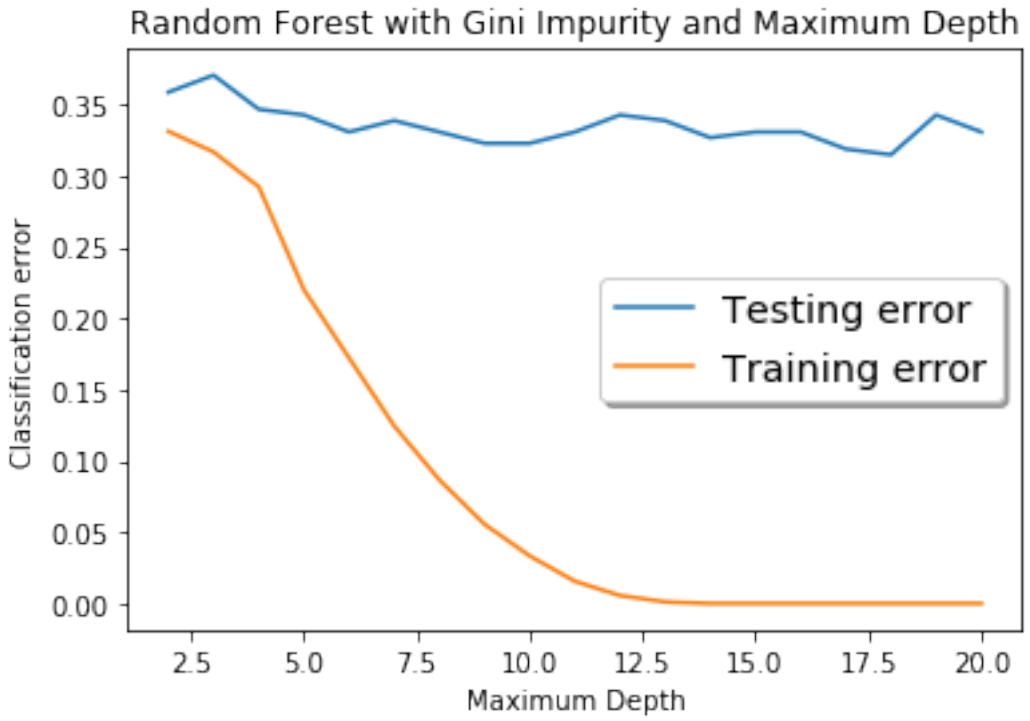



```
Test error minimized at min_samples_leaf = 19
```

1.4 Problem 2E: Random Forests with Maximum Depth Stopping Criterion

[8]:
```
# Seed the random number generator:
np.random.seed(1)

clf = RandomForestClassifier(n_estimators = n_estimators, criterion = 'gini')

max_depth = np.arange(2, 21)

train_err, test_err = eval_tree_based_model_max_depth(clf, max_depth,␣
 ↪train_data,
                                                       train_label, test_data,␣
 ↪test_label)

plt.figure()
```

```
plt.plot(max_depth, test_err, label='Testing error')
plt.plot(max_depth, train_err, label='Training error')
plt.xlabel('Maximum Depth')
plt.ylabel('Classification error')
plt.title('Random Forest with Gini Impurity and Maximum Depth')
plt.legend(loc=0, shadow=True, fontsize='x-large')
plt.show()

print('Test error minimized at max_depth = %i' % max_depth[np.argmin(test_err)])
```




```
Test error minimized at max_depth = 18
```

3_notebook

January 30, 2020

1 Problem 3

In this Jupyter notebook, we visualize how boosting and AdaBoost work. This notebook requires FFmpeg; instructions to install it can be found in set 1.

Use this notebook to write your code for problem 3.

[1]:
```
# Setup.

import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from IPython.display import HTML

from boosting_helper import (
    generate_dataset,
    visualize_dataset,
    gb_suite, ab_suite,
    visualize_loss_curves_gb, visualize_loss_curves_ab,
    animate_gb, animate_ab
)
```

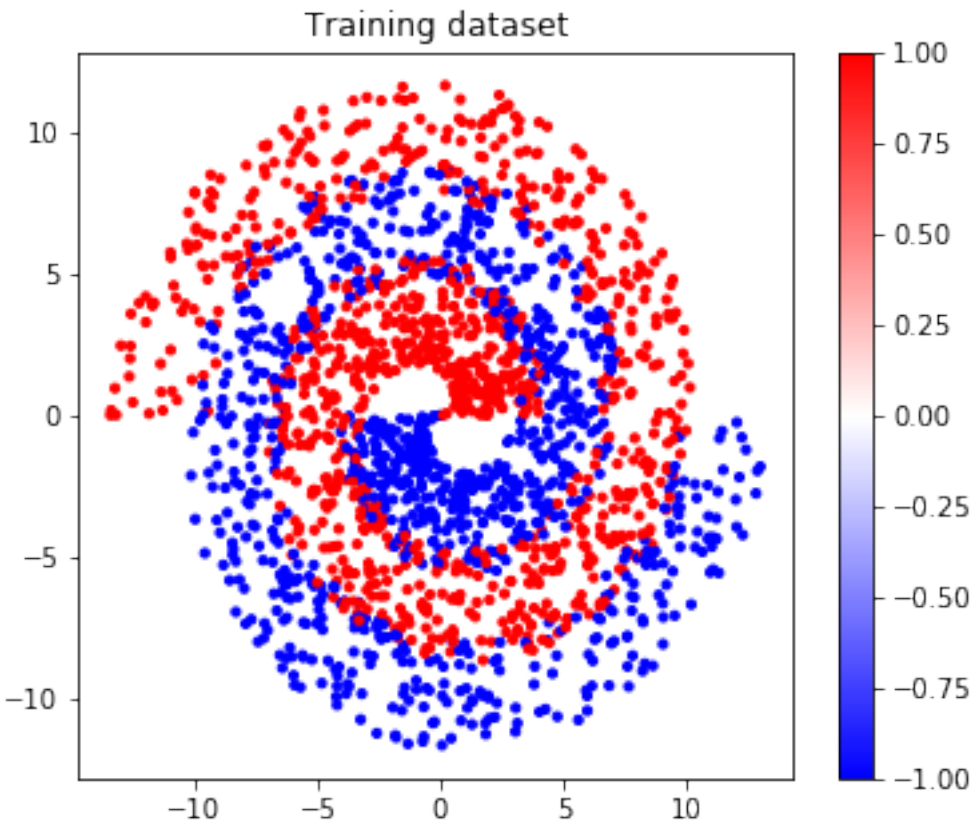
1.1 Dataset

We'll start off by generating a complex, slightly noisy 2-dimensional dataset (namely, two spirals) with +1 or -1 as labels. Note that learning on this dataset is a classification problem.

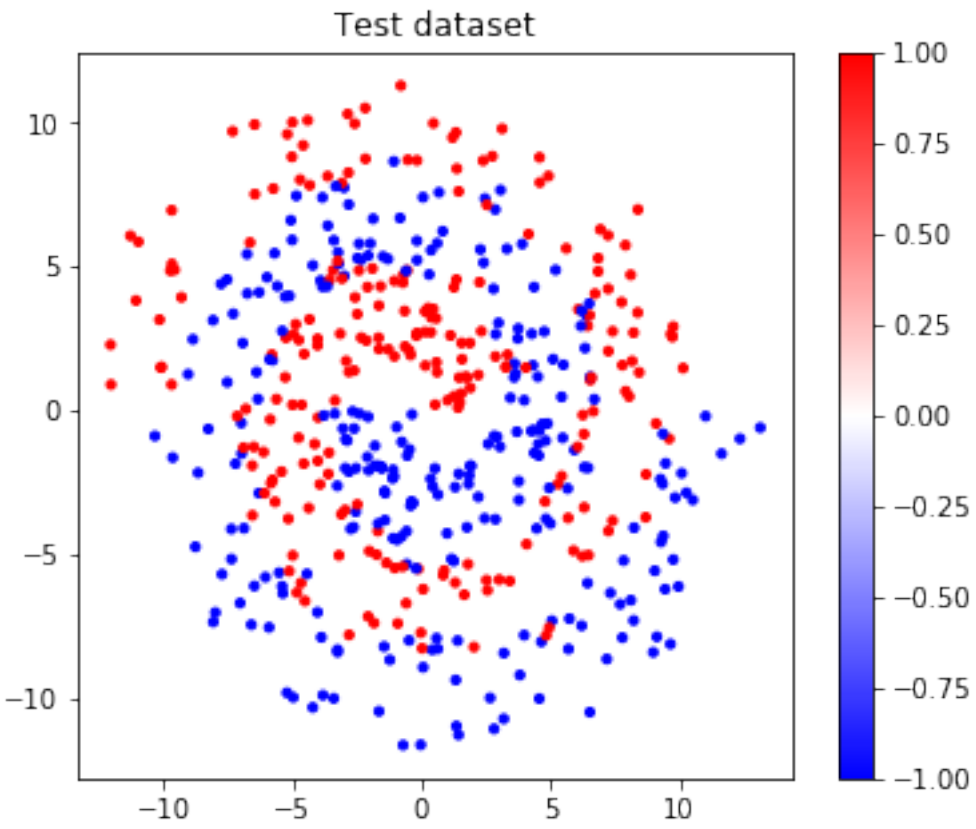
(Note: red indicates positive and blue indicates negative.)

[2]:
```
# Generate a dataset with 2000 training points and 500 test points.
# Refer to the source code for more details.
(X_train, Y_train), (X_test, Y_test) = generate_dataset(2000, 500, 1.5, 4.0)

# Visualize the generated dataset.
visualize_dataset(X_train, Y_train, 'Training dataset')
visualize_dataset(X_test, Y_test, 'Test dataset')
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



1.2 Gradient Boosting

Let's implement a simple gradient boosting model to classify this dataset. Since we are implementing gradient boosting, we will need to use regressors even though we are dealing with a classification problem. To resolve this issue, we can simply take the sign of the predictions as the predictions of the classifier.

Our weak regressors will be decision trees with a maximum of `n_nodes=4` leaf nodes. You can use the following line to create a DT weak regressor:

```
clf = DecisionTreeRegressor(max_leaf_nodes=n_nodes)
```

Fill in the `fit()` method in the cell below.

[3]:
```
class GradientBoosting():
    def __init__(self, n_clfs=100):
        '''
        Initialize the gradient boosting model.

        Inputs:
            n_clfs (default 100): Initializer for self.n_clfs.

        Attributes:
            self.n_clfs: The number of DT weak regressors.
```

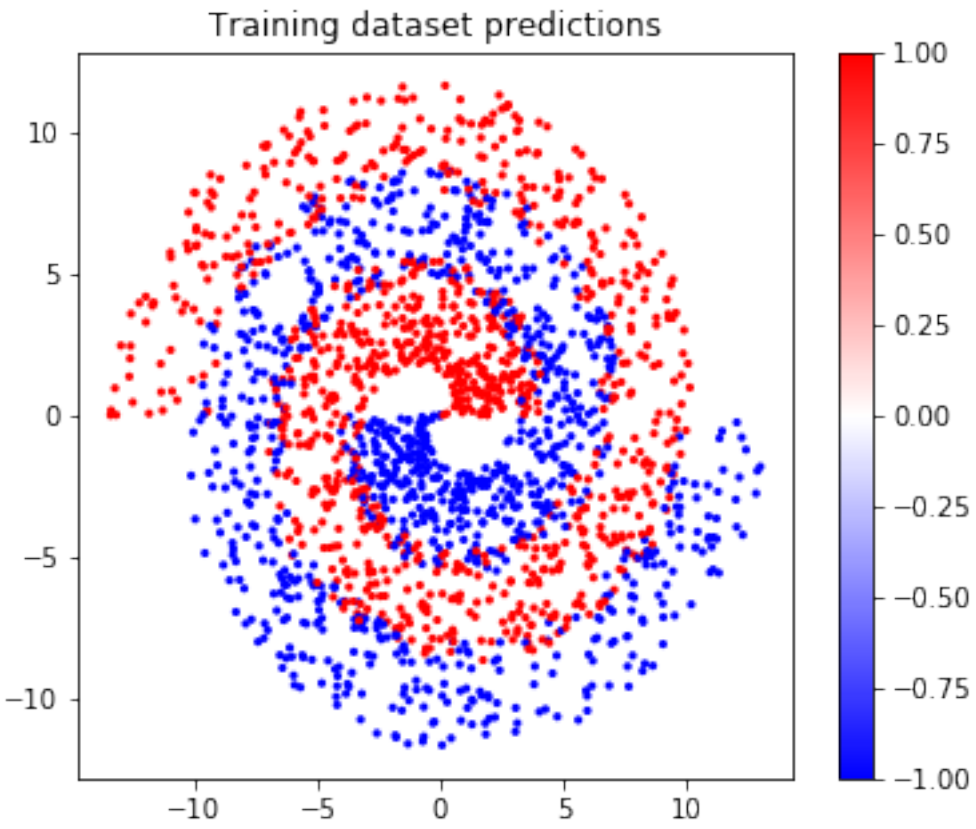
```
            self.clfs: A list of the DT weak regressors, initialized as empty.
        '''
        self.n_clfs = n_clfs
        self.clfs = []

    def fit(self, X, Y, n_nodes=4):
        '''
        Fit the gradient boosting model. Note that since we are implementing this method in a class,
        rather than having a bunch of inputs and outputs, you will deal with the attributes of the class.
        (see the __init__() method).

        This method should thus train self.n_clfs DT weak regressors and store them in self.clfs.

        Inputs:
            X: A (N, D) shaped numpy array containing the data points.
            Y: A (N, ) shaped numpy array containing the (float) labels of the data points.
                (Even though the labels are ints, we treat them as floats.)
            n_nodes: The max number of nodes that the DT weak regressors are allowed to have.
        '''
        Y_train = Y
        for i in range(self.n_clfs):
            clf = DecisionTreeRegressor(max_leaf_nodes=n_nodes)
            clf.fit(X, Y_train)
            self.clfs.append(clf)
            Y_train = Y_train - clf.predict(X)

        pass

    def predict(self, X):
        '''
        Predict on the given dataset.

        Inputs:
            X: A (N, D) shaped numpy array containing the data points.

        Outputs:
            A (N, ) shaped numpy array containing the (float) labels of the data points.
            (Even though the labels are ints, we treat them as floats.)
        '''
        # Initialize predictions.
```

```
        Y_pred = np.zeros(len(X))

        # Add predictions from each DT weak regressor.
        for clf in self.clfs:
            Y_curr = clf.predict(X)
            Y_pred += Y_curr

        # Return the sign of the predictions.
        return np.sign(Y_pred)

    def loss(self, X, Y):
        '''
        Calculate the classification loss.

        Inputs:
            X: A (N, D) shaped numpy array containing the data points.
            Y: A (N, ) shaped numpy array containing the (float) labels of the␣
↪data points.
                (Even though the labels are ints, we treat them as floats.)

        Outputs:
            The classification loss.
        '''
        # Calculate the points where the predictions and the ground truths␣
↪don't match.
        Y_pred = self.predict(X)
        misclassified = np.where(Y_pred != Y)[0]

        # Return the fraction of such points.
        return float(len(misclassified)) / len(X)
```

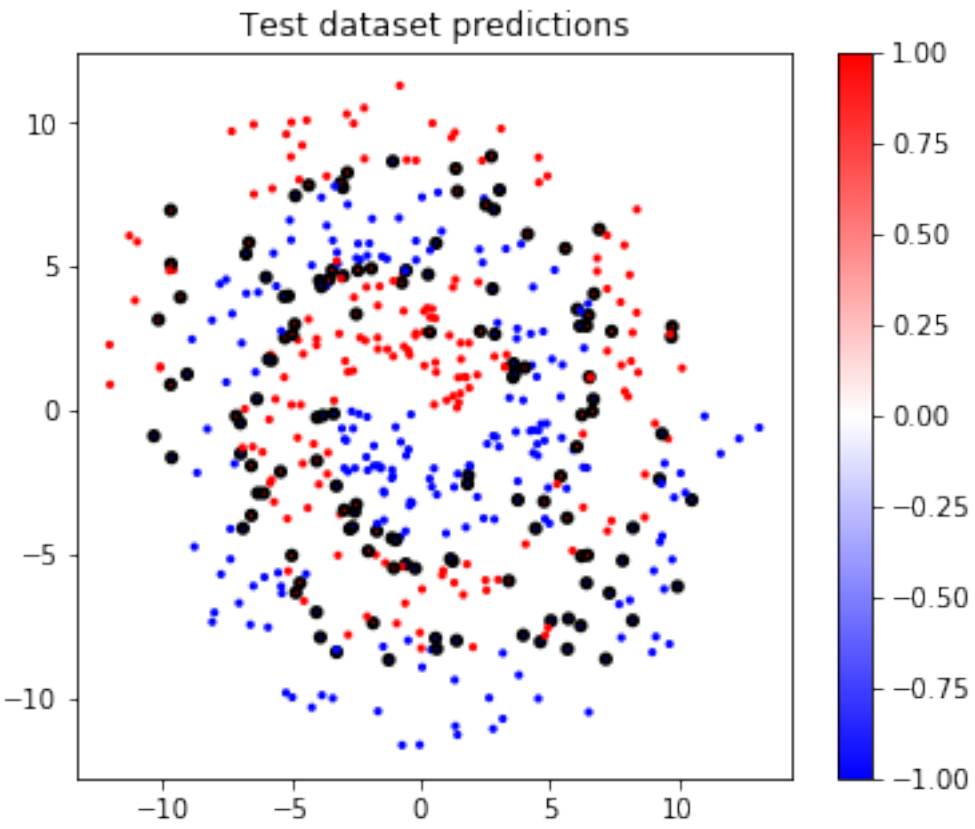
Let's plot the prediction results with 500 weak regressors. Note that misclassified points are marked in black.

[4]:
```
model = gb_suite(GradientBoosting, 500, X_train, Y_train, X_test, Y_test)
```

```
<Figure size 432x288 with 0 Axes>
```




```
Training loss: 0.000000

<Figure size 432x288 with 0 Axes>
```




```
Test loss: 0.264000
```

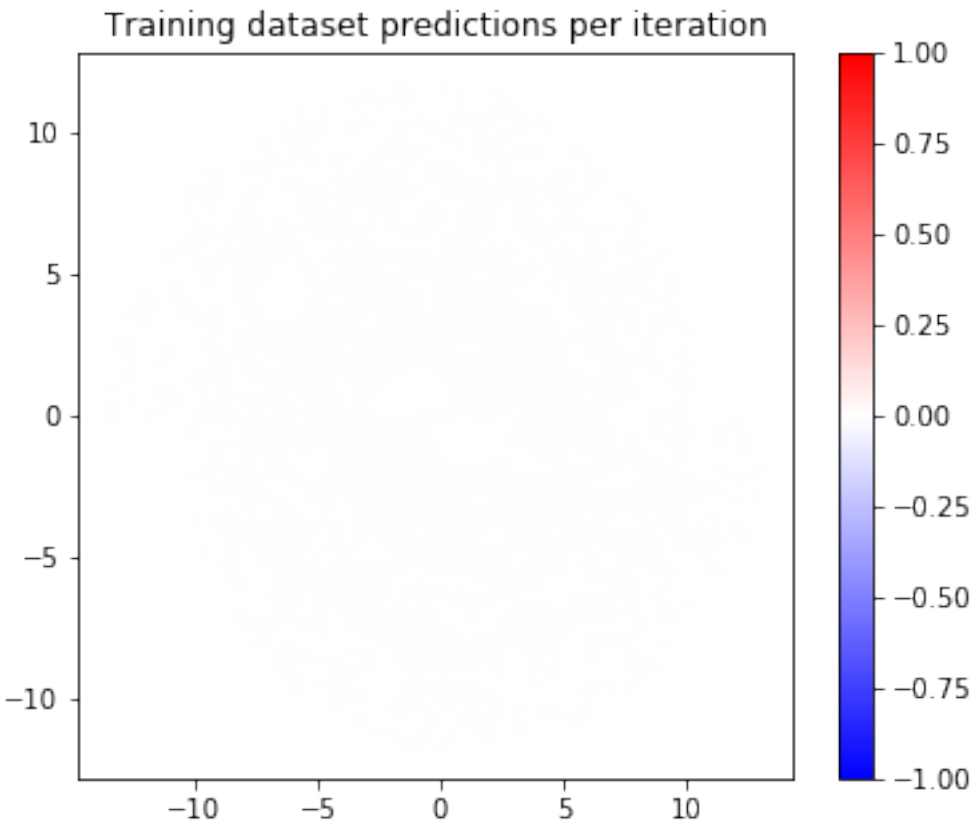
1.3 Visualization of Gradient Boosting Training

Let's visualize the training process of the gradient boosting model. First, we visualize how the predictions change with each new weak regressor that we train and add:

[5]:
```
anim = animate_gb(model, X_train, Y_train, 'Training dataset predictions per␣
↪iteration')
HTML(anim.to_html5_video())
```

```
Animating...
```

[5]: `<IPython.core.display.HTML object>`



Now, we visualize how the loss decreases with each new weak regressor:

[6]:
```
visualize_loss_curves_gb(model, X_train, Y_train, X_test, Y_test)
```

1.4 AdaBoost

We used regression above for a classification problem. A better approach would be to use AdaBoost, which is a natural adaptation for classification.

Let's implement an AdaBoost model to classify this dataset. This time, our weak classifiers (not regressors!) will be decision trees with a maximum of `n_nodes=4` leaf nodes. You can use the following line to create a DT weak classifier:

`clf = DecisionTreeClassifier(max_leaf_nodes=n_nodes)`

Fill in the fit() method in the cell below.

NOTE: Use the 0/1 loss here instead of the exponential loss!

[7]:
```
class AdaBoost():
    def __init__(self, n_clfs=100):
        '''
        Initialize the AdaBoost model.

        Inputs:
            n_clfs (default 100): Initializer for self.n_clfs.

        Attributes:
            self.n_clfs: The number of DT weak classifiers.
            self.coefs: A list of the AdaBoost coefficients.
            self.clfs: A list of the DT weak classifiers, initialized as empty.
        '''
```

```
        self.n_clfs = n_clfs
        self.coefs = []
        self.clfs = []

    def fit(self, X, Y, n_nodes=4):
        '''
        Fit the AdaBoost model. Note that since we are implementing this method in a class, rather
        than having a bunch of inputs and outputs, you will deal with the attributes of the class.
        (see the __init__() method).

        This method should thus train self.n_clfs DT weak classifiers and store them in self.clfs,
        with their coefficients in self.coefs.

        Inputs:
            X: A (N, D) shaped numpy array containing the data points.
            Y: A (N, ) shaped numpy array containing the (float) labels of the data points.
                (Even though the labels are ints, we treat them as floats.)
            n_nodes: The max number of nodes that the DT weak classifiers are allowed to have.

        Outputs:
            A (N, T) shaped numpy array, where T is the number of iterations / DT weak classifiers,
            such that the t^th column contains D_{t+1} (the dataset weights at iteration t+1).
        '''
        Y_train = Y
        D = np.array([]) # (N, T) shaped numpy array, weight of training data at each iteration
        D_t = np.ones(len(X)) / len(X)

        for i in range(self.n_clfs):
            clf = DecisionTreeClassifier(max_leaf_nodes=n_nodes)
            clf.fit(X, Y_train, sample_weight = D_t) # train weak classifier
            self.clfs.append(clf)

            epsilon = 0
            for j in np.where(clf.predict(X) != Y)[0]: # find places where loss exists
                epsilon += np.dot(D_t[j], 1) # calculate 0/1 loss

            # determine weight of weak classifier
```

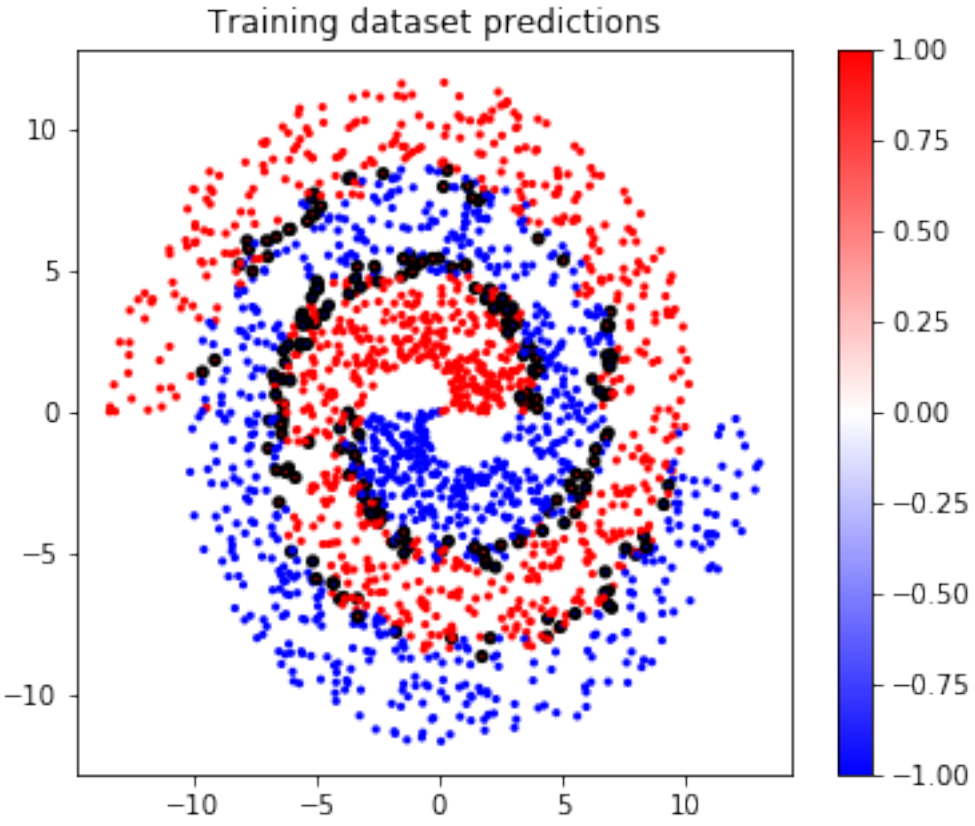
```
            alpha_t = 0.5 * np.log((1 - epsilon) / epsilon)
            self.coefs.append(alpha_t)

            D = np.append(D, D_t)
            # update weight of training data
            Z_t = (1 - epsilon) * np.exp(- alpha_t) + epsilon * np.exp(alpha_t)
            D_t = D_t * np.exp(-alpha_t * Y * clf.predict(X)) / Z_t

        return D.reshape((len(X), self.n_clfs))

        pass


    def predict(self, X):
        '''
        Predict on the given dataset.

        Inputs:
            X: A (N, D) shaped numpy array containing the data points.

        Outputs:
            A (N, ) shaped numpy array containing the (float) labels of the␣
↪data points.
            (Even though the labels are ints, we treat them as floats.)
        '''
        # Initialize predictions.
        Y_pred = np.zeros(len(X))

        # Add predictions from each DT weak classifier.
        for i, clf in enumerate(self.clfs):
            Y_curr = self.coefs[i] * clf.predict(X)
            Y_pred += Y_curr

        # Return the sign of the predictions.
        return np.sign(Y_pred)

    def loss(self, X, Y):
        '''
        Calculate the classification loss.

        Inputs:
            X: A (N, D) shaped numpy array containing the data points.
            Y: A (N, ) shaped numpy array containing the (float) labels of the␣
↪data points.
                (Even though the labels are ints, we treat them as floats.)

        Outputs:
```

```
            The classification loss.
        '''
        # Calculate the points where the predictions and the ground truths␣
↪don't match.
        Y_pred = self.predict(X)
        misclassified = np.where(Y_pred != Y)[0]

        # Return the fraction of such points.
        return float(len(misclassified)) / len(X)
```

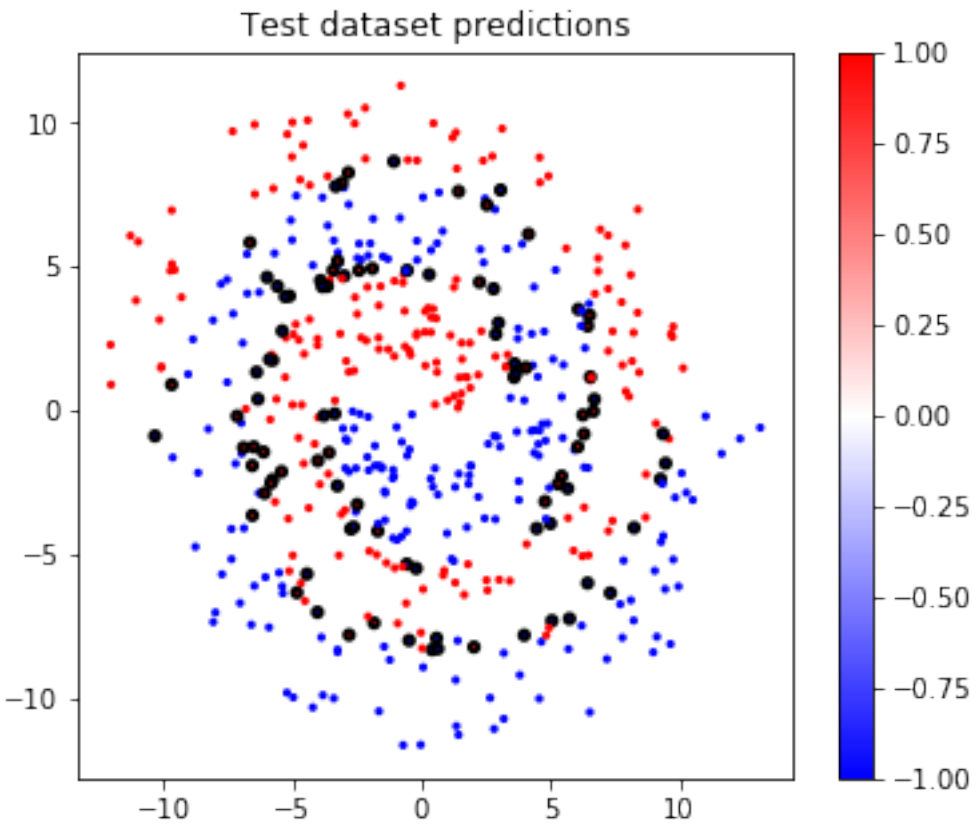
Again, let's plot the prediction results with 500 weak classifiers (misclassified points marked in black).

[8]:
```
model, D = ab_suite(AdaBoost, 500, X_train, Y_train, X_test, Y_test)
```

```
<Figure size 432x288 with 0 Axes>
```




```
Training loss: 0.104500

<Figure size 432x288 with 0 Axes>
```




```
Test loss: 0.186000
```

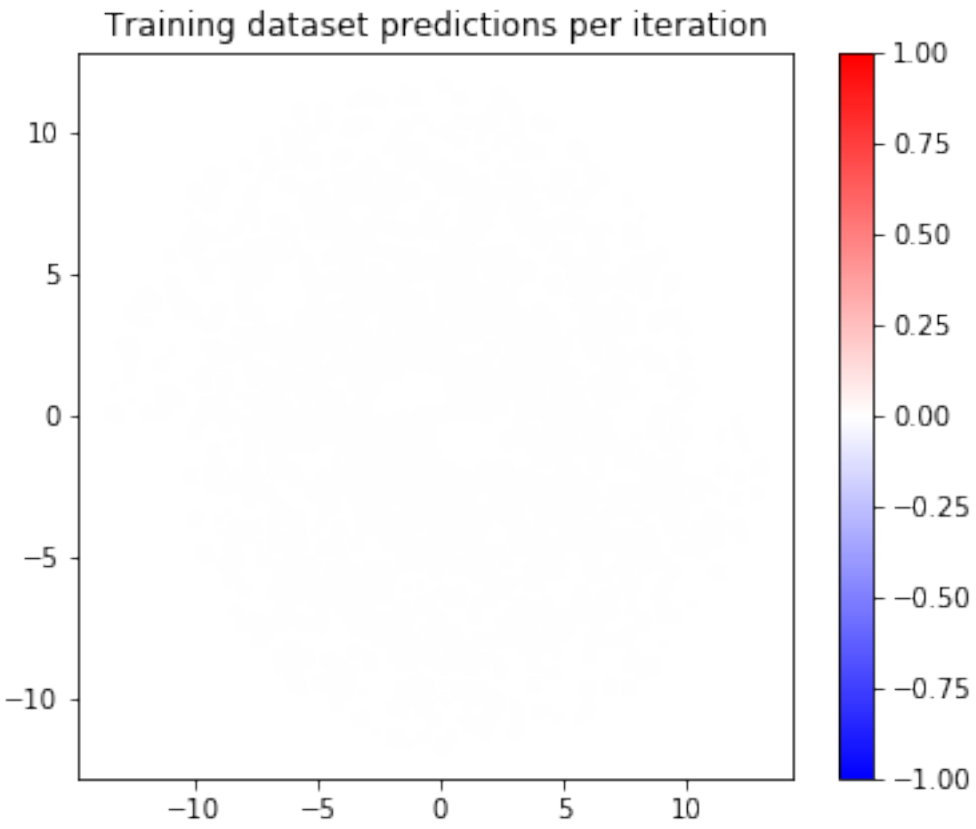
1.5 Visualization of AdaBoost Training

Let's now visualize the training process of the AdaBoost model. First, we visualize how the predictions, as well as the dataset weights, change with each new weak classifier that we train and add:

[9]:
```
anim = animate_ab(model, X_train, Y_train, D, 'Training dataset predictions per␣
↪iteration')
HTML(anim.to_html5_video())
```

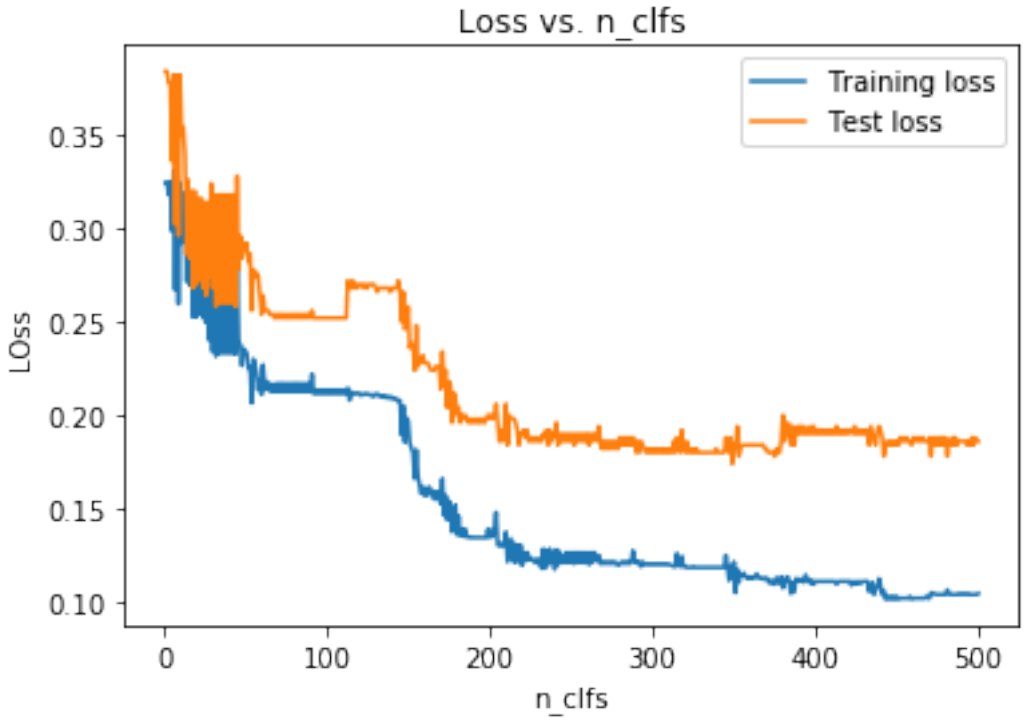
```
Animating...
```

[9]: `<IPython.core.display.HTML object>`



Now, we visualize how the loss decreases with each new weak classifier:

[10]:
```
visualize_loss_curves_ab(model, X_train, Y_train, X_test, Y_test)
```


Loss vs. n_clfs
Training loss
Test loss
LOss
0.35
0.30
0.25
0.20
0.15
0.10
0
100
200
300
400
500
n_clfs