Homework 4

Kyle Hadley

In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
import warnings
warnings.simplefilter('ignore')
```

1. PCA via Successive Deflation

Note: Worked problem with Charlie and Joaquin from class on this problem.

(a)

For the covariance of the deflated matrix, we are given the relationship $\tilde{X}^T = (I - v_1 v_1^T) X^T$. From this relationship, we know that $\tilde{X} = X(I - v_1 v_1^T)$.

From these relationships, we start with our covariance of the deflated matrix such that,

$$\tilde{C} = \frac{1}{n} \tilde{X}^T \tilde{X}$$

Substituting our relationships for $\tilde{X}^T$ and $\tilde{X}$,

$$\tilde{C} = \frac{1}{n} (I - v_1 v_1^T) X^T X (I - v_1 v_1^T)$$

We can substitute $\frac{1}{n} X^T X = C$ such that

$$\tilde{C} = (I - v_1 v_1^T) C (I - v_1 v_1^T)$$

$$\tilde{C} = C - v_1 v_1^T C - C v_1 v_1^T + v_1 v_1^T C v_1 v_1^T$$

Substituing our relationship $Cv_1 = \lambda_1 v_1$ and given that λ is a constant,

$$\tilde{C} = C - v_1 v_1^T C - \lambda_1 v_1 v_1^T + v_1 v_1^T \lambda_1 v_1 v_1^T$$

$$\tilde{C} = C - v_1 v_1^T C - \lambda_1 v_1 v_1^T + \lambda_1 v_1 v_1^T v_1 v_1^T$$

Also, given our orthonormality relationship we know that $v_1^T v_1 = 1$; thus,

$$\tilde{C} = C - v_1 v_1^T C - \lambda_1 v_1 v_1^T + \lambda_1 v_1 v_1^T$$

$$\tilde{C} = C - v_1 v_1^T C$$

We can take a "double" transpose of our left-most term (i.e. $a = (a^T)^T$); thus,

$$\tilde{C} = C - ((v_1 v_1^T C)^T)^T$$

$$\tilde{C} = C - (C^T v_1 v_1^T)^T$$

From our relationship for C, we can see that $C = C^T$ ($C = \frac{1}{n} X^T X = C^T$); thus,

$$\tilde{C} = C - (C v_1 v_1^T)^T$$

Applying our relationship again of $Cv_1 = \lambda_1 v_1$ and $C = \frac{1}{n} X^T X$,

$$\tilde{C} = \frac{1}{n} X^T X - (\lambda_1 v_1 v_1^T)^T$$

$$\tilde{C} = \frac{1}{n} X^T X - \lambda_1 v_1 v_1^T$$

(b)

For a $j \neq 1$ and v_j as a principle eigvenvector of C, we can show that v_j is also a principal eigenvector of $\tilde{C}$ with the same eigenvalue λ_j.

Given $\tilde{C} = \frac{1}{n} X^T X - \lambda_1 v_1 v_1^T$, we right-multiply both sides by v_j,

$$\tilde{C} v_j = \left(\frac{1}{n} X^T X - \lambda_1 v_1 v_1^T \right) v_j$$

$$\tilde{C} v_j = \frac{1}{n} X^T X v_j - \lambda_1 v_1 v_1^T v_j$$

Given our orthonormality relationship, we know that if $j \neq 1$ then $v_1^T v_j = 0$; thus,

$$\tilde{C} v_j = \frac{1}{n} X^T X v_j - \lambda_1 v_1 (0)$$

$$\tilde{C} v_j = \frac{1}{n} X^T X v_j$$

Substituting $\frac{1}{n} X^T X = C$,

$$\tilde{C} v_j = C v_j = \lambda_j v_j$$

Thus, we can see that v_j is also a principal eigenvector of $\tilde{C}$ with the same eigenvalue λ_j.

(c)

By deflating our matrix C to $\tilde{C}$, we are essentially removing one of the parameters from our initial dataset as determined by the principal eigenvector - which was given as v_1. Since we know that we have removed the parameter associated with v_1 when creating our deflated covariance matrix, we know that v_1 is no longer the first principal eigenvector of $\tilde{C}$. Thus, v_2 must be the first principal eigenvector for $\tilde{C}$.

(d)

Intuitively, when we multiply our vector u_0 by our covariance matrix C, we end up stretching our u_0 along the eigenvectors of C. However, the stretching is greatest along the principal eigenvector direction.

Thus, if we repeat this process of stretching for a large number of times (k times), we will get a resulting u_k will look like a stretched version of our principal eigenvector. We can then see that u_k is just a scaled version of the principal eigenvector - thus meaning that they are equivalent.

The computational complexity is $O(m^2k)$ as we are performing a matrix multiplication of C by itself (which is a $O(m)$ operation) k times.

It would be advantageous to include the norming component as referenced in eq. (5) because this avoids our values increasing or decreasing exponentially as we stretch u_0. This ensures that our value is within the memory limits of computers.

(e)

In [6]:

```
def calculate_eigens(C, k):
    n, m = C.shape

    u0 = np.ones(n)
    u = np.linalg.matrix_power(C, k).dot(u0)

    u = u / np.linalg.norm(u)

    lamda = C.dot(u)[0] / u[0]
    return lamda, u.T

def calculate_K_eigens(C, X, K, k):
    n, m = C.shape

    lamda = np.zeros(K)
    v = np.zeros(shape=(n, K))

    for i in range(K):
        lamda[i], v[:, i] = calculate_eigens(C, k)
        u = np.expand_dims(v[:, i], axis=1)
        X = X.dot(np.identity(n) - u.dot(u.T))
        C = 1/n * (X.T).dot(X)

    #print(C)

    return lamda, v

n = 10
m = 5
np.random.seed(0)
X = np.random.randint(3, size=(n, m))
#print(X)
C = 1/n * (X.T).dot(X)

l, v = calculate_K_eigens(C, X, 5, 50)
l_true, v_true = np.linalg.eig(C)
```

```
print('Eigenvalues:\n', 'Predicted:', l, '\n', 'Actual:', l_true)
print('\nEigenvectors:\n', 'Predicted:\n', v, '\n', 'Actual:\n', v_true)
```

```
Eigenvalues:
 Predicted: [4.14066235 4.0241905  0.89120969 0.55592762 0.04734749]
 Actual: [4.14066235 2.01209525 0.02367374 0.27796381 0.44560484]

Eigenvectors:
 Predicted:
 [[ 0.42723111 -0.51084397  0.15881388  0.61225951  0.39551039]
 [ 0.3670085   0.52528157  0.12882228  0.50999197 -0.5591932 ]
 [ 0.59423189 -0.41956691  0.1213297  -0.51070685 -0.44193855]
 [ 0.33912602  0.47707352  0.54570421 -0.31125022  0.51256669]
 [ 0.46331452  0.24388967 -0.80353531 -0.08572222  0.26988923]]
 Actual:
 [[-0.42723111 -0.51084397  0.39551039  0.61225951 -0.15881388]
 [-0.3670085   0.52528157 -0.5591932   0.50999197 -0.12882228]
 [-0.59423189 -0.41956691 -0.44193855 -0.51070685 -0.1213297 ]
 [-0.33912602  0.47707352  0.51256669 -0.31125022 -0.54570421]
 [-0.46331452  0.24388967  0.26988923 -0.08572222  0.80353531]]
```

2. Locality Sensitive Hashing

(a)

In order to avoid the 3rd condition where the magical oracle is unstable, we set $c = 1$ such that $r = cr$. This will ensure that we avoid the unstable condition when requesting a nearest neighbor.

After setting $c = 1$, we can perform iterative queries by starting with an arbitrary r_0. Given an r_0 we can ask the oracle for the nearest neighbor of a query point q.

If the oracle returns a point x', then we will reduce our r_0 by half (rounded down) and perform a new query with our new r_1 (i.e. half the original r_0). If the oracle still returns a point x', then we continue to reduce our r_i by half (as long as it's possible with integer values) until we've reached a point where the oracle finds nothing. Once the oracle finds nothing, then we take the average of our r_{i-1} from the previous iteration and the latest r_i for r_{i+1}. We perform this "yo-yo"-ing until we reach a point where we know that there is no smaller r that yields an x'.

We perform a similar series of steps if the oracle doesn't return a point x' with our initial r_0 but instead set $r_{i+1} = 2r_i$ until the oracle does return an x'. Then we begin the same "yo-yo"ing as described above until we reach a set r.

(b)

The lower bound for the probability that h maps x_i and x_j to the same value is a function of the number of common values within x_i and x_j at the same indices. While we don't have a function or value for the common values, we can use the Hamming distance as this represents where x_i and x_j have differing values at various positions within the vectors.

Given that $d(x_i, x_j) \leq r$, we know that the lower bound for common values (i.e. $h(x_i) = h(x_j)$) is when $d(x_i, x_j) = r$; thus, we can describe the probability of $h(x_i) = h(x_j)$ as,

$$p_1 = Pr(h(x_i) = h(x_j)) = 1 - \frac{r}{m}$$

where m is the size of the binary vectors x_i and x_j.

Given $d(x_i, x_j) \geq r$, we know that upper bound for common values is when $d(x_i, x_j) = cr$; thus, we can describe the probability of $h(x_i) = h(x_j)$ as,

$$p_2 = Pr(h(x_i) = h(x_j)) = 1 - \frac{cr}{m}$$

The inequality relationship between p_1 and p_2 is $p_2 \leq p_1$.

(c)

Given that $g(x_i) = (h_1(x_i), h_2(x_i), \ldots, h_k(x_i))$, we can surmise that the probability of $g(x_i) = g(x_j)$ is equivalent to the combined probability of $h_1(x_i) = h_1(x_j)$, $h_2(x_i) = h_2(x_j)$, ..., $h_k(x_i) = h_k(x_j)$; thus,

$$Pr(g(x_i) = g(x_j)) = Pr(h_1(x_i) = h_1(x_j)) \times Pr(h_2(x_i) = h_2(x_j)) \times \ldots \times Pr(h_k(x_i) = h_k(x_j))$$

Substituting our p_1 for our lower bound condition, the lower bound of our probability is

$$Pr(g(x_i) = g(x_j)) = p_1 \times p_1 \times \ldots \times p_1$$

$$Pr(g(x_i) = g(x_j)) = p_1^k$$

Substituting our p_2 for our lower bound condition, the lower bound of our probability is

$$Pr(g(x_i) = g(x_j)) = p_2 \times p_2 \times \ldots \times p_2$$

$$Pr(g(x_i) = g(x_j)) = p_2^k$$

(d)

To find the probability that for $b \in \{0, 1, \ldots, l-1\}$ where $g_b(x_i) = g_b(x_j)$, we can consider the probability that this will never occur such that,

$$Pr\left(\exists b \in \{0, 1, \ldots, l-1\} : g_b(x_i) = g_b(x_j)\right) = 1 - Pr\left(g_b(x_i) \neq g_b(x_j), \forall b\right)$$

We know from part (c) that the probability of $Pr(g(x_i) = g(x_j)) = p_1^k$ for lower and upper bounds. Thus, we know that for $Pr(g(x_i) \neq g(x_j)) = 1 - p_1^k$. Given that we are calculating the probability across all values of b, we can then surmise that,

$$Pr\left(g_b(x_i) \neq g_b(x_j), \forall b\right) = (1 - p_1^k)^l$$

Thus, for the lower bound we know that

$$Pr\left(\exists b \in \{0, 1, \ldots, l-1\} : g_b(x_i) = g_b(x_j)\right) = 1 - (1 - p_1^k)^l$$

Similarly, for the upper bound we know that

$$Pr\left(\exists b \in \{0, 1, \ldots, l-1\} : g_b(x_i) = g_b(x_j)\right) = 1 - (1 - p_2^k)^l$$

(e)

Given that $\rho = \frac{\ln(p_1)}{\ln(p_2)}$, $l = n^{\rho}$, and $k = \frac{\ln(n)}{\ln(1/p_2)}$.

Solving the probability of the first event, given that $d(x', q) \leq r$ we know that we are dealing with the lower bound probability as defined in part (d). Thus, we know that,

$$Pr(A) = 1 - (1 - p_1^k)^l$$

where $P(A)$ is the probability of the first event. First substituting our relationship for k and l, we find that

$$Pr(A) = 1 - \left(1 - p_1^{\frac{\ln(n)}{\ln(1/p_2)}}\right)^{n^{\rho}}$$

Examining the term $p_1^{\frac{ln(n)}{\ln(1/p_2)}}$, we can re-write this as

$$p_1^{\ln(n)^{\frac{1}{\ln(1/p_2)}}}$$

We can apply the log base change rule such that $\ln(n) = \frac{\log_{p_1} n}{\log_{p_1} e}$; thus,

$$p_1^{\frac{\log_{p_1} n}{\log_{p_1} e}^{\frac{1}{\ln(1/p_2)}}}$$

We know that $p_1^{\log_{p_1} n} = n$; thus,

$$n^{\frac{1}{\log_{p_1} e}^{\frac{1}{\ln(1/p_2)}}}$$

Again, applying the log base change rule it is known that $\frac{1}{\log_{p_1}(e)} = \frac{\ln(p_1)}{\ln(e)} = \ln(p_1)$; thus,

$$n^{\ln(p_1)^{\frac{1}{\ln(1/p_2)}}} = n^{\frac{\ln(p_1)}{\ln(1/p_2)}} = \frac{1}{n^{\frac{\ln(p_1)}{\ln(p_2)}}}$$

We know from our relationship for ρ that,

$$\frac{1}{n^{\frac{\ln(p_1)}{\ln(p_2)}}} = \frac{1}{n^{\rho}}$$

Re-plugging this new value into our initial equation for $Pr(A)$ we see that,

$$Pr(A) = 1 - \left(1 - \frac{1}{n^{\rho}}\right)^{n^{\rho}}$$

As highlighted by the hint in the homework, we know that for an arbitrary value, z, the $\lim_{k \to \infty}(1 - 1/z)^z = e^{-1}$ and that $(1 - 1/z)^z$ will always be less than e^{-1}. Thus, we can surmise

that the $Pr(A)$ can be re-written as,

$$Pr(A) \geq 1 - e^{-1}$$

Thus, we see that the first event happens with probability of at least $1 - e^{-1}$.

Solving the probability of the second event, we can define our probability as,

$$Pr(B) = Pr(W \leq 4l)$$

where $Pr(B)$ is the probability of the second event, W is our random variable of the number of items in X. We can re-write this such that

$$Pr(B) = 1 - Pr(W \geq 4l)$$

From this relationship, we apply Markov's inequality such that,

$$Pr(B) \geq 1 - \frac{E[W]}{4l}$$

where $E(W)$ represents the expectation of W defined as $E[W] = \sum x_i p_i$. The probability in this relationship is $p_i = l * Pr(g_b(x) = g_b(q))$ for a given b, as defined by part (c), p^k. We know that we are dealing with the upper limit, i.e. p_2, because $d(x, q) \geq cr$. Thus, we first simplifying our $p_i = l * p_2^k$,

$$p_i = l * p_2^k = l * p_2^{\frac{\ln(n)}{\ln(1/p_2)}} = l * p_2^{-\frac{\ln(n)}{\ln(p_2)}}$$

We can apply the log base change rule such that $\frac{\ln(n)}{\ln(p_2)} = \log_{p_2}(n)$; thus,

$$l * p_2^{-\frac{\ln(n)}{\ln(p_2)}} = l * p_2^{-\log_{p_2}(n)} = l * n^{-1} = l * \frac{1}{n}$$

Thus, we know that probability of there being a single x and q where $p_i = l * \frac{1}{n}$. We know that this probability is equivalent for all values of W; thus, we can re-write our expectation as,

$$E[W] = \sum x_i p_i = np = n\left(\frac{l}{n}\right) = l$$

Plugging this value into our $Pr(B)$ equation, we see that,

$$Pr(B) \geq 1 - \frac{l}{4l}$$

$$Pr(B) \geq \frac{3}{4}$$

Thus, we see that the probability of the second event is at least $3/4$.

The lower bound on the probability of both events happening is defined by

$$Pr(A \cap B) \geq Pr(A) * Pr(B)$$

using the lower bound of both 1st and 2nd events. Thus,

$$Pr(A \cap B) \geq \frac{3}{4}(1 - e^{-1})$$

(f)

3. Programming Problem: Random Forests

Note: Worked on this problem with Niraj and Charlie from class on problems 3 and 4.

(a)

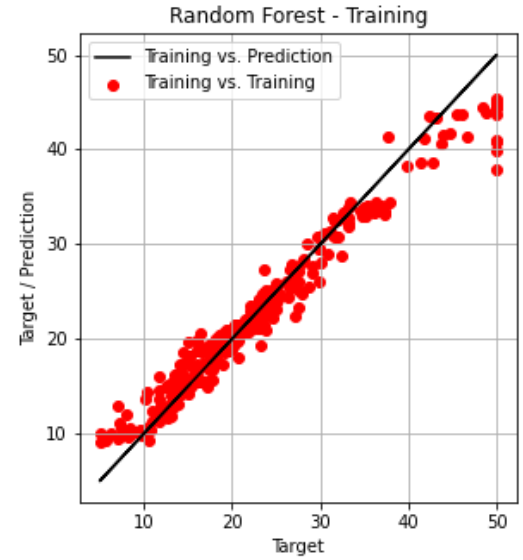
Reference *GBDT.ipynb* for coding implementation of Random Forest code.

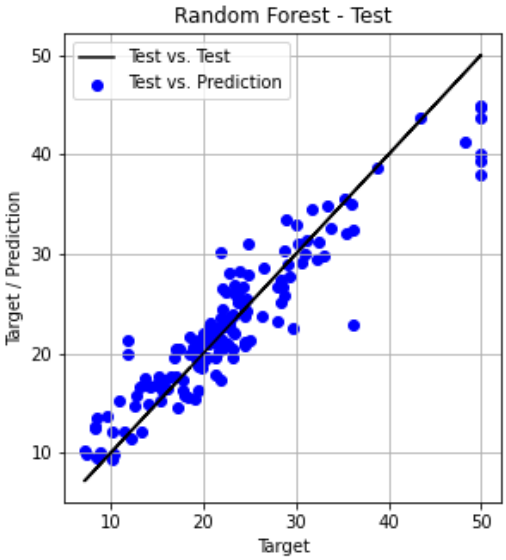
Given the Boston house price dataset, below in the table is the calculated training and test RMSE using the Random Forest object. In addition, the table also included the results from least square regression and ridge regression as calculated in previous homeworks.

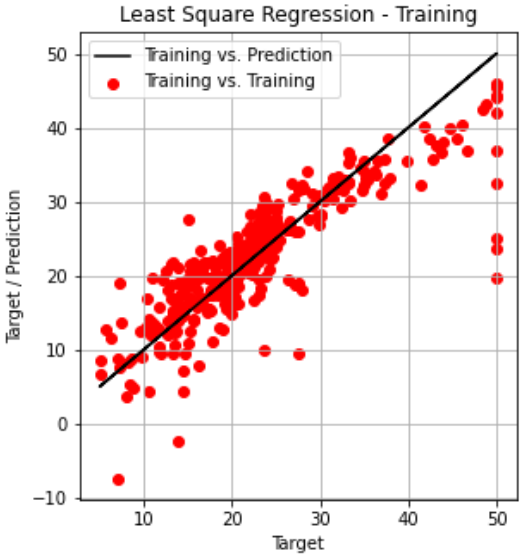
ML Technique	RMSE (Training Data)	RMSE (Test Data)
Random Forest (RF)	2.131	3.343
Least Square Regression	4.8206	5.2092
Ridge Regression	4.8419	5.1474

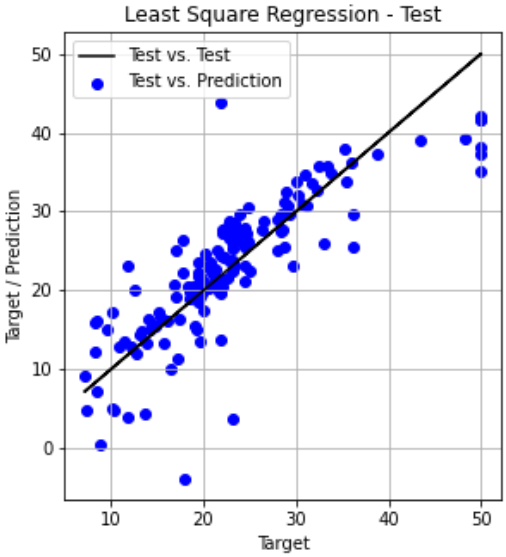
We can see that the Random Forest performs significantly better than both least square and ridge regression based on RMSE score for the training dataset and performs better for the test dataset.

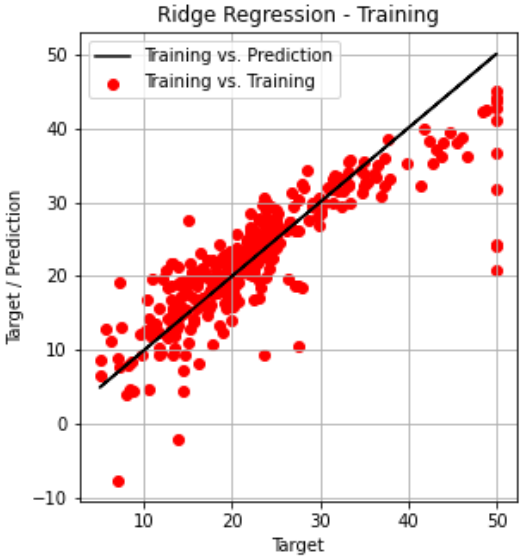
The plots below demonstrate the performance of RF vs. the other regression solutions. The plots below demonstrate how close the predicted value is to the actual value for both our training and test datasets. The line in each graph represents a perfect correlation (i.e. if prediction was equal to actual) and the scattered dots represent the predictions with respect to actuals.

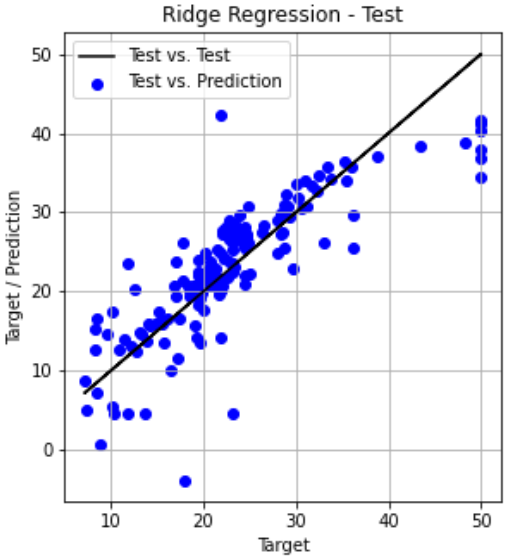

Random Forest - Training
Training vs. Prediction
Training vs. Training
Target / Prediction
Target


Random Forest - Test
Test vs. Test
Test vs. Prediction
Target / Prediction
Target


Least Square Regression - Training
Training vs. Prediction
Training vs. Training
Target / Prediction
Target


Least Square Regression - Test
Test vs. Test
Test vs. Prediction
Target / Prediction
Target


Ridge Regression - Training
Training vs. Prediction
Training vs. Training
Target / Prediction
Target


Ridge Regression - Test
Test vs. Test
Test vs. Prediction
Target / Prediction
Target

(b)

Reference *GBDT.ipynb* for coding implementation of Random Forest code.

Given the Credit-g and Brest Cancer Diagnostic datasets, below in the table is the calculated training and test accuracy using the Random Forest object.

Dataset	Accuracy (Training Data)	Accuracy (Test Data)
Credit-g	90.71 %	78.33 %
Brest Cancer Diagnostic	98.99 %	97.66 %

4. Programming Problem: Gradient Boosting Decision Trees

(a)

The computational complexity of optimizing a tree of depth d in terms of m and n is $O(nmd)$.

(b)

It seems that the most computational expensive part of the GBDT training is the building of each individual tree.

Besides performing parallel programming, we could attempt to optimize the tree building by identifying which features are most likely to drive the division of training data. For example, we may know that feature j has only two values - thus, we can infer that our dataset should be split along this feature immediately (assuming that feature j has a non-negligble impact on our calculated value).

There is also a technique called decision tree pruning in which we remove from the decision tre sections of the tree that are non-critical and/or redundant. For example, we may perform a split on our root node by feature j and then perform a split on the left child again by feature j. We may be able to prune some of the nodes from the tree that exhibit this type of behavior.

(c)

The recursive calling of left and right child can be performed in parallel as the generation of the children nodes have independent inputs / outputs from each other. Thus, we can be generating the left child (and subsequent children) in parallel to the right child.

(d)

One way in which GBDT is different from other gradient descent algorithims and approaches is the way in which the model updates itself. With gradient descent, we descend the differentiable loss function by introducing changes to our parameters (the weights we calculate). For GBDT, rather than changing the parameters we instead descend the gradient by introducing new models - the is similar to updating our weights but slightly different.

Another difference is the way in which GBDT updates the "parameters" (i.e. the way it adds a model). Gradient descent calculates it updates with respect to each individual value of our training set while GBDT updates with respect to a very small subset of our training set at each leaf node (dependent on the value of our *min_sample_split*).

(e)

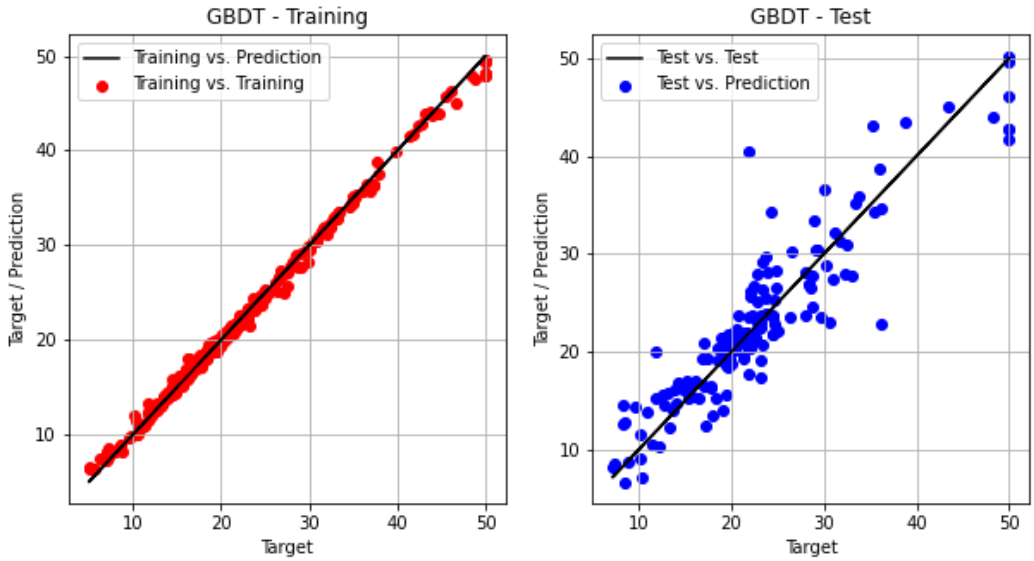
Reference *GBDT.ipynb* for coding implementation of Random Forest code.

Given the Boston house price dataset, below in the table is the calculated training and test RMSE using the GBDT object. In addition, the table also included the results from least square regression and ridge regression as calculated in previous homeworks.

ML Technique	RMSE (Training Data)	RMSE (Test Data)
GBDT	0.568	3.597
Least Square Regression	4.8206	5.2092
Ridge Regression	4.8419	5.1474

We can see that the GBDT performs significantly better than both least square and ridge regression based on RMSE score for the training dataset and performs better for the test dataset. The extremely low RMSE value for the training data, but larger RMSE for test data, indicates that the model may be tuned too much to the training data. However, the model as captured in code seems to optimize the test data RMSE as best as possible (barring small optimization tweaks).

The plot below demonstrates the performance of GBDT vs. the other regression solutions. The plot demonstrates how close the predicted value is to the actual value for both our training and test datasets. The line in each graph represents a perfect correlation (i.e. if prediction was equal to actual) and the scattered dots represent the predictions with respect to actuals. This plot can be compared with the other plots for least square and ridge regression and presented in 3(a).



(f)

Reference *GBDT.ipynb* for coding implementation of Random Forest code.

Given the Credit-g and Brest Cancer Diagnostic datasets, below in the table is the calculated training and test accuracy using the GDBT object.

Dataset	Accuracy (Training Data)	Accuracy (Test Data)
Credit-g	96.57 %	78.00 %
Brest Cancer Diagnostic	99.75 %	95.32 %

(g)

It seems that across all three experiments that GBDT was signficantly more accurate for the training dataset classification or calculation, but was roughly equivalent to the accuracy for the test dataset.

A few possible explanations:

- The inputted parameters for both RF and GDBT caused the models to overfit to the training datasets. I attempted to optimize my models based on test dataset accuracy, without significantly causing lower accuracy on my training dataset (i.e. fudging the numbers to get a good test dataset accuracy).
- The size of the dataset may be indicative of which model may prove more accurate. If GDBT was more effective at classifying or predicting for my training dataset, it seems that having a larger dataset that I can train on would help the model be more indicative of my real dataset (outside of my training/test data).

GBDT.ipynb

In [1]:

```
#from multiprocessing import Pool
#from functools import partial
import matplotlib.pyplot as plt
import numpy as np
import time
#from numba import jit

import warnings
warnings.simplefilter('ignore')
```

In [2]:

```
'''
    "score" = predicted value, should be single value
    pred() takes GBDT/RF outputs, i.e., the "score", as its inputs, and returns predict
    g() is the gradient/1st order derivative, which takes true values "true" and scores
    h() is the heassian/2nd order derivative, which takes true values "true" and scores
'''
class leastsquare(object):
    '''Loss class for mse. As for mse, pred function is pred=score.'''
    def pred(self, score):
        return score

    def g(self, true, score):
        #score_mat = np.ones(shape=true.shape) * score
        return -2*(true - score)

    def h(self, true, score):
        return 2*np.ones(shape=true.shape)

class logistic(object):
    '''Loss class for log loss. As for log loss, pred function is logistic transformati
    def pred(self, score):
        prediction = 1 / (1 + np.exp(-score))
        prediction[prediction > 0.5] = 1
        prediction[prediction <= 0.5] = 0
        return prediction

    def g(self, true, score):
        if score.shape != true.shape:
            score_mat = np.ones(shape=true.shape) * score
            return (-(np.exp(score_mat)*(true - 1) + true) / (np.exp(score_mat) + 1))
        else:
            return (-(np.exp(score)*(true - 1) + true) / (np.exp(score) + 1))

    def h(self, true, score):
        if score.shape != true.shape:
            score_mat = np.ones(shape=true.shape) * score
            return np.exp(score_mat) / (np.exp(score_mat) + 1)**2
        else:
            return np.exp(score) / (np.exp(score) + 1)**2
```

In [3]:

```
# TODO: class of GBDT
class GBDT(object):
    '''
```

```
    Class of gradient boosting decision tree (GBDT)

    Parameters:
        n_threads: The number of threads used for fitting and predicting.
        loss: Loss function for gradient boosting.
            'mse' for regression task and 'log' for classfication task.
            A child class of the loss class could be passed to implement customized los
        max_depth: The maximum depth D_max of a tree.
        min_sample_split: The minimum number of samples required to further split a nod
        lamda: The regularization coefficient for leaf score, also known as lambda.
        gamma: The regularization coefficient for number of tree nodes, also know as ga
        learning_rate: The learning rate eta of GBDT.
        num_trees: Number of trees.
    '''
    def __init__(self, n_threads = None, loss = 'mse', max_depth = 3, min_sample_split

        self.n_threads = n_threads

        if loss == 'mse':
            self.loss = leastsquare()
        elif loss == 'log':
            self.loss = logistic()
        else:
            print('Invalid loss function for RF object.')
            return self

        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.learning_rate = learning_rate
        self.num_trees = num_trees
        self.trees = []
        self.init_pred = 0

    def fit(self, train, target):
        # train is n x m 2d numpy array
        # target is n-dim 1d array

         # First, find our initial prediction y_0 which is the average of our inputted
        self.init_pred = np.mean(target)
        y_pred = np.ones(shape=target.shape) * self.init_pred

        g = self.loss.g(target, y_pred)
        h = self.loss.h(target, y_pred)

        # Generate self.num_trees
        for i in np.arange(self.num_trees):

            new_tree = Tree(n_threads = self.n_threads, max_depth = self.max_depth, min
            new_tree.fit(train=train, target=target, g=g, h=h)
            self.trees.append(new_tree)

            # Update our values for g and h based on the previously created tree; our n
            y_pred = self.learning_rate * new_tree.predict(train) + y_pred

            g = self.loss.g(target, y_pred)
            h = self.loss.h(target, y_pred)
        return self

    def predict(self, test):
```

```
        n, m = test.shape

        score = np.ones(n) * self.init_pred

        for tree in self.trees:
            score += tree.predict(test) * self.learning_rate

        return self.loss.pred(score)

    def print(self):
        for tree in self.trees:
            tree.print()
        return self
```

In [4]:

```
# TODO: class of Random Forest
class RF(object):
    '''
    Class of Random Forest

    Parameters:
        n_threads: The number of threads used for fitting and predicting.
        loss: Loss function for gradient boosting.
            'mse' for regression task and 'log' for classfication task.
            A child class of the loss class could be passed to implement customized los
        max_depth: The maximum depth d_max of a tree.
        min_sample_split: The minimum number of samples required to further split a nod
        lamda: The regularization coefficient for leaf score, also known as lambda.
        gamma: The regularization coefficient for number of tree nodes, also know as ga
        rf: rf*m is the size of random subset of features, from which we select the bes
        num_trees: Number of trees.
    '''
    def __init__(self, n_threads = None, loss = 'mse', max_depth = 3, min_sample_split

        self.n_threads = n_threads

        if loss == 'mse':
            self.loss = leastsquare()
        elif loss == 'log':
            self.loss = logistic()
        else:
            print('Invalid loss function for RF object.')
            return self

        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.rf = rf
        self.num_trees = num_trees
        self.trees = []
        self.init_pred = 0

    def fit(self, train, target):
        # train is n x m 2d numpy array
        # target is n-dim 1d array

        # First, find our initial prediction y_0 which is the average of our inputted t
        self.init_pred = np.mean(target)
```

```
        g = self.loss.g(target, self.init_pred)
        h = self.loss.h(target, self.init_pred)
        #print(g, h)

        # Generate self.num_trees
        for i in np.arange(self.num_trees):
            train_boot = np.random.choice(train.shape[0], size=train.shape[0], replace=

            new_tree = Tree(n_threads = self.n_threads, max_depth = self.max_depth, min
            new_tree.fit(train=train[train_boot, :], target=target[train_boot], g=g[tra
            self.trees.append(new_tree)

        return self

    def predict(self, test):
        n, m = test.shape

        score = np.zeros(n)

        for tree in self.trees:
            score += tree.predict(test)

        score = score / self.num_trees
        score += self.init_pred

        return self.loss.pred(score)

    def print(self):
        for tree in self.trees:
            tree.print()
        return self
```

In [5]:

```
# TODO: class of a node on a tree
class TreeNode(object):
    '''
    Data structure that are used for storing a node on a tree.

    A tree is presented by a set of nested TreeNodes,
    with one TreeNode pointing two child TreeNodes,
    until a tree leaf is reached.

    A node on a tree can be either a leaf node or a non-leaf node.
    '''

    def __init__(self, y_value = 0, split_feature = None, split_threshold = None, left_
        #[X1, X2, index_y, value_y] = split(X)
        if not left_child and not right_child:
            self.is_leaf = True
        else:
            self.is_leaf = False
        self.y_value = y_value
        self.left_child = left_child
        self.right_child = right_child
        self.split_feature = split_feature
        self.split_threshold = split_threshold
        self.gain = gain

    def forward(self, x):
        if x[self.split_feature] < self.split_threshold:
```

```
            return self.left_child
        else:
            return self.right_child

    def print(self):
        print('Is Leaf:', self.is_leaf, '| Value:', self.y_value, '| Split feature:', s

        if(self.is_leaf == False):
            self.left_child.print()
            self.right_child.print()

        return self
```

In [6]:

```
# TODO: class of single tree
class Tree(object):
    '''
    Class of a single decision tree in GBDT

    Parameters:
        n_threads: The number of threads used for fitting and predicting.
        max_depth: The maximum depth of the tree.
        min_sample_split: The minimum number of samples required to further split a nod
        lamda: The regularization coefficient for leaf prediction, also known as lambda
        gamma: The regularization coefficient for number of TreeNode, also know as gamm
        rf: rf*m is the size of random subset of features, from which we select the bes
            rf = 0 means we are training a GBDT.
    '''

    def __init__(self, n_threads = None, max_depth = 3, min_sample_split = 10, lamda =
        self.n_threads = n_threads
        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.rf = rf
        self.int_member = 0
        self.root_node = None

    def print(self):
        self.root_node.print()

        return self

    def fit(self, train, target, g, h):
        '''
        train is the training data matrix, and must be numpy array (an n_train x m matr
        g and h are gradient and hessian respectively.
        '''
        n, m = train.shape

        self.root_node = self.construct_tree(train, target, g, h, 1)

        return self

    def predict(self, test):
        '''
        test is the test data matrix, and must be numpy arrays (an n_test x m matrix).
        Return predictions (scores) as an array.
        '''
```

```
        n, m = test.shape
        result = np.zeros(n)

        for i in np.arange(n):
            current_node = self.root_node
            test_current = test[i, :]

            while(current_node.is_leaf != True):
                current_node = current_node.forward(test_current)

            result[i] = current_node.y_value

        return result

    def construct_tree(self, train, target, g, h, current_depth):
        '''
        Tree construction, which is recursively used to grow a tree.
        First we should check if we should stop further splitting.

        The stopping conditions include:
            1. tree reaches max_depth $d_{max}$
            2. The number of sample points at current node is less than min_sample_spli
            3. gain <= 0
        '''
        n, m = train.shape
        #print('Iteration Size:', n)
        #print('construct_tree | size of train:', train.shape)

        if current_depth == self.max_depth or n <= self.min_sample_split:
            # Return a leaf node where the value of the leaf node defined by eq. 19
            y_value = -(np.sum(g)) / (np.sum(h) + self.lamda)
            #print('Leaf Node Size:', n, y_value)
            return TreeNode(y_value=y_value)
        else:
            # Set current node as non-leaf node and create children leaf nodes
            feature, threshold, gain = self.find_best_decision_rule(train, g, h)
            L = train[:, feature] < threshold # array of true/false
            R = train[:, feature] >= threshold

            #print(current_depth, train[L].shape, train[R].shape, threshold)

            # Check for condition where splitting results in a leaf node of value 0
            if gain < 0 or train[L].shape[0] == 0 or train[R].shape[0] == 0:
                y_value = -(np.sum(g)) / (np.sum(h) + self.lamda)
                #print('Leaf Node Size:', n, y_value)
                return TreeNode(y_value=y_value)

            left_child = self.construct_tree(train[L], target[L], g[L], h[L], current_d
            right_child = self.construct_tree(train[R], target[R], g[R], h[R], current_


        return TreeNode(split_feature = feature, split_threshold = threshold, left_chil

    def find_best_decision_rule(self, train, g, h):
        '''
        Return the best decision rule [feature, treshold], i.e., $(p_j, \tau_j)$ on a n
        train is the training data assigned to node j
        g and h are the corresponding 1st and 2nd derivatives for each data point in tr
        g and h should be vectors of the same length as the number of data points in tr

        for each feature, we find the best threshold by find_threshold(),
```

```
        a [threshold, best_gain] list is returned for each feature.
        Then we select the feature with the largest best_gain,
        and return the best decision rule [feature, treshold] together with its gain.
        '''
        n, m = train.shape

        # Calculate which features we'll be evaluating for best threshold+gain - this i
        if self.rf != 0:
            #print(m, self.rf)
            m_prime = np.random.choice(m, size=(int(np.round_(m*self.rf))), replace=Fal
        else:
            m_prime = np.arange(m)

        m_prime = np.sort(m_prime)

        threshold_arr = np.zeros(m_prime.size)
        best_gain_arr = np.zeros(m_prime.size)

        #print('fit | calculated m_prime:', m_prime)

        for i in np.arange(m_prime.size):
            #print(m_prime[i])
            threshold_arr[i], best_gain_arr[i] = self.find_threshold(g, h, train[:, m_p

        best_gain = np.amax(best_gain_arr)
        threshold = threshold_arr[np.argmax(best_gain_arr)]
        feature = m_prime[np.argmax(best_gain_arr)]

        #print('find_best_decision_rule | return:', threshold, best_gain, feature)

        return feature, threshold, best_gain

    def find_threshold(self, g, h, train):
        '''
        Given a particular feature $p_j$,
        return the best split threshold $\tau_j$ together with the gain that is achieve
        '''
        # Assume that train is a [n x 1] matrix containing data only about the particul
        # Sort our train matrix in ascending fashion
        n = train.size
        sort_i = np.argsort(train)
        threshold_arr = np.zeros(n - 1)
        best_gain_arr = np.zeros(n - 1)

        for i in np.arange(start = 0, stop = (n - 1)):
            threshold_arr[i] = (train[sort_i[i+1]] + train[sort_i[i]])/2

            # Split train into 2 sets - Left and Right
            L = (train < threshold_arr[i])
            R = (train >= threshold_arr[i])

            #print(threshold_arr[i], train[L], train[R])
            G_L = np.sum(g[L])
            G_R = np.sum(g[R])
            H_L = np.sum(h[L])
            H_R = np.sum(h[R])

            best_gain_arr[i] = 1/2 * ((G_L**2 / (H_L + self.lamda)) + (G_R**2/ (H_R + s

        # Return the largest gain and its associated threshold
        best_gain = np.amax(best_gain_arr)
```

```
        #print(best_gain_arr)
        threshold = threshold_arr[np.argmax(best_gain_arr)]

        #print('find_threshold | return:', [threshold, best_gain])
        return threshold, best_gain
```

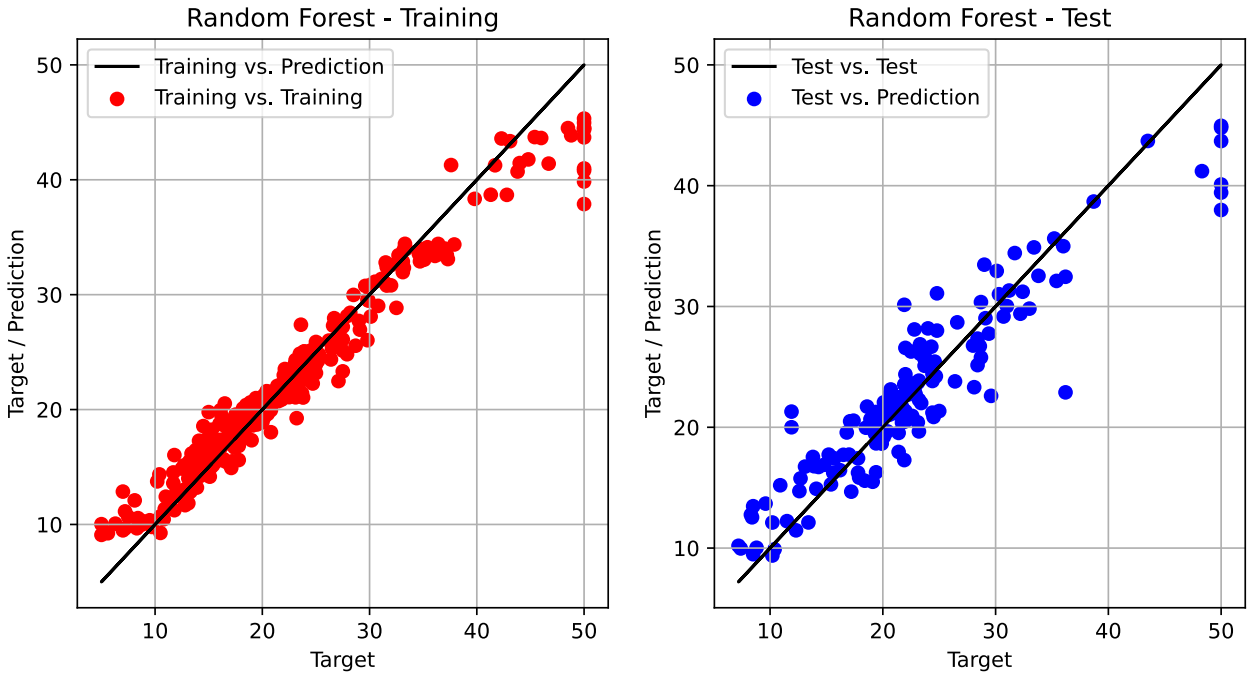
In [7]:

```
# TODO: Evaluation functions (you can use code from previous homeworks)

# RMSE
def root_mean_square_error(pred, y):
    rmse = np.sqrt(np.mean((pred - y)**2))
    return rmse

# precision
def accuracy(pred, y):
    precision = np.sum(y == pred) / np.size(y)
    return precision
```

In [35]:

```
# load data
from sklearn import datasets
boston = datasets.load_boston()
X = boston.data
y = boston.target

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=8

# Test the performance of RF on the Boston house price dataset
boston_RF = RF(loss = 'mse', max_depth = 10, min_sample_split = 1, lamda = 1, gamma = 1

boston_RF.fit(X_train, y_train)

y_pred_train = boston_RF.predict(X_train)
y_pred_test = boston_RF.predict(X_test)

print('Boston RF RMSE (train):', root_mean_square_error(y_pred_train, y_train))
print('Boston RF RMSE (test):', root_mean_square_error(y_pred_test, y_test))

fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].scatter(y_train, y_pred_train, c='r', label='Training vs. Training')
ax[0].plot(y_train, y_train, 'k-', label='Training vs. Prediction')
ax[0].grid()
ax[0].legend()
ax[0].set_xlabel('Target')
ax[0].set_ylabel('Target / Prediction')
ax[0].set_title('Random Forest - Training')

ax[1].scatter(y_test, y_pred_test, c='b', label='Test vs. Prediction')
ax[1].plot(y_test, y_test, 'k-', label='Test vs. Test')
ax[1].grid()
ax[1].legend()
ax[1].set_xlabel('Target')
ax[1].set_ylabel('Target / Prediction')
ax[1].set_title('Random Forest - Test')
fig.show()
plt.savefig('Problem_3_a_1')
```

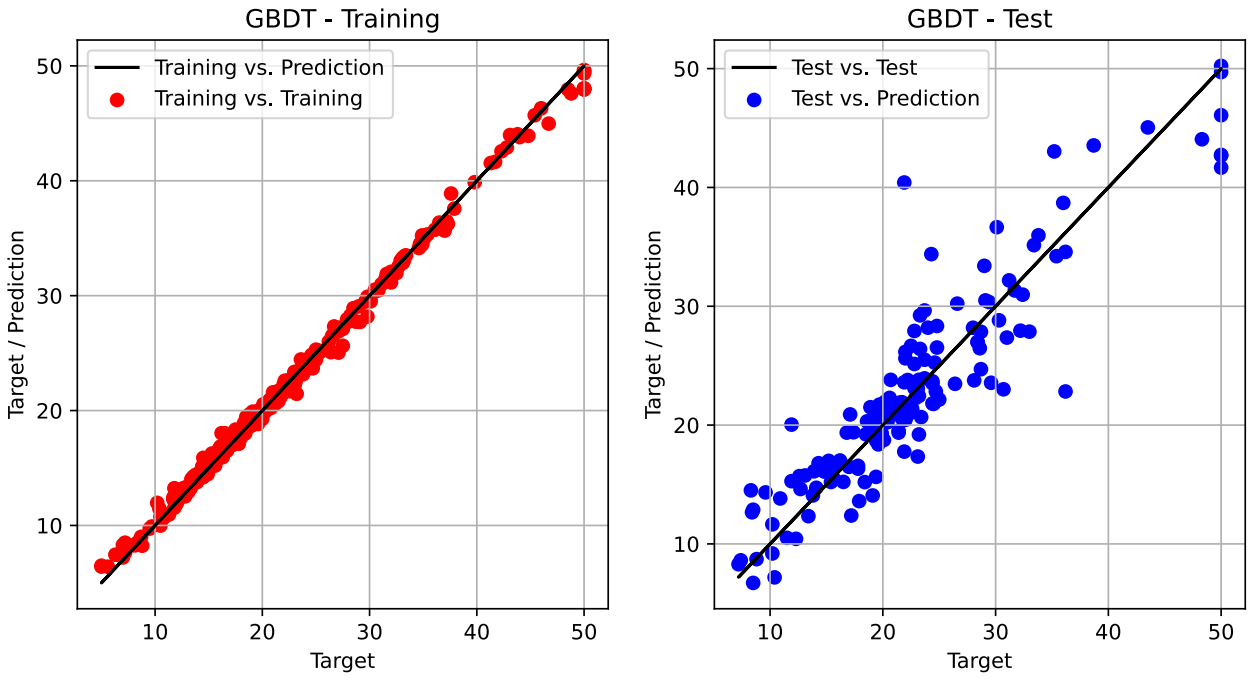
```
Boston RF RMSE (train): 2.1309253469877296
Boston RF RMSE (test): 3.3431306455576153
```



In [59]:

```
# Test the performance of GBDT on the Boston house price dataset
boston_GBDT = GBDT(loss = 'mse', max_depth = 10, min_sample_split = 10, lamda = 5, gamm

boston_GBDT.fit(X_train, y_train)

y_pred_train = boston_GBDT.predict(X_train)
y_pred_test = boston_GBDT.predict(X_test)

print('Boston GBDT RMSE (train):', root_mean_square_error(y_pred_train, y_train))
print('Boston GBDT RMSE (test):', root_mean_square_error(y_pred_test, y_test))

fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].scatter(y_train, y_pred_train, c='r', label='Training vs. Training')
ax[0].plot(y_train, y_train, 'k-', label='Training vs. Prediction')
ax[0].grid()
ax[0].legend()
ax[0].set_xlabel('Target')
ax[0].set_ylabel('Target / Prediction')
ax[0].set_title('GBDT - Training')

ax[1].scatter(y_test, y_pred_test, c='b', label='Test vs. Prediction')
ax[1].plot(y_test, y_test, 'k-', label='Test vs. Test')
ax[1].grid()
ax[1].legend()
ax[1].set_xlabel('Target')
ax[1].set_ylabel('Target / Prediction')
ax[1].set_title('GBDT - Test')
fig.show()
plt.savefig('Problem_4_e_1')
```

```
Boston GBDT RMSE (train): 0.5676113232637662
Boston GBDT RMSE (test): 3.5966095296367127
```



In [76]:

```
# Load data
from sklearn.datasets import fetch_openml
X, y = fetch_openml('credit-g', version=1, return_X_y=True, data_home='credit/', as_fra
y = np.array(list(map(lambda x: 1 if x == 'good' else 0, y)))

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=8

# Test the performance of RF on the credit-g dataset
credit_RF = RF(loss = 'log', max_depth = 10, min_sample_split = 10, lamda = 1, gamma =

credit_RF.fit(X_train, y_train)

y_pred_train = credit_RF.predict(X_train)
y_pred_test = credit_RF.predict(X_test)

print('Credit-g RF Accuracy (train):', accuracy(y_pred_train, y_train)*100, '%')
print('Credit-g RF Accuracy (test):', accuracy(y_pred_test, y_test)*100, '%')
```

```
Credit-g RF Accuracy (train): 90.71428571428571 %
Credit-g RF Accuracy (test): 78.33333333333333 %
```

In [77]:

```
# Test the performance of GBDT on the credit-g dataset
credit_GBDT = GBDT(loss = 'log', max_depth = 10, min_sample_split = 10, lamda = 8, gamm

credit_GBDT.fit(X_train, y_train)

y_pred_train = credit_GBDT.predict(X_train)
y_pred_test = credit_GBDT.predict(X_test)

print('Credit-g GBDT Accuracy (train):', accuracy(y_pred_train, y_train)*100, '%')
print('Credit-g GBDT Accuracy (test):', accuracy(y_pred_test, y_test)*100, '%')
```

```
Credit-g GBDT Accuracy (train): 96.57142857142857 %
Credit-g GBDT Accuracy (test): 78.0 %
```

In [80]:
```
# TODO: GBDT classification on breast cancer dataset

# load data
from sklearn import datasets
breast_cancer = datasets.load_breast_cancer()
X = breast_cancer.data
y = breast_cancer.target

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=8

# First, test the performance of RF on the credit-g dataset
cancer_RF = RF(loss = 'log', max_depth = 10, min_sample_split = 5, lamda = 1, gamma = 0

cancer_RF.fit(X_train, y_train)

y_pred_train = cancer_RF.predict(X_train)
y_pred_test = cancer_RF.predict(X_test)

print('Breast Cancer RF Accuracy (train):', accuracy(y_pred_train, y_train)*100, '%')
print('Breast Cancer RF Accuracy (test):', accuracy(y_pred_test, y_test)*100, '%')
```

```
Breast Cancer RF Accuracy (train): 98.99497487437185 %
Breast Cancer RF Accuracy (test): 97.6608187134503 %
```

In [82]:
```
# Test the performance of GBDT on the credit-g dataset
#cancer_GBDT = GBDT(loss = 'log', max_depth = 5, min_sample_split = 5, lamda = 1, gamma
cancer_GBDT = GBDT(loss = 'log', max_depth = 10, min_sample_split = 5, lamda = 5, gamma

cancer_GBDT.fit(X_train, y_train)

y_pred_train = cancer_GBDT.predict(X_train)
y_pred_test = cancer_GBDT.predict(X_test)

print('Breast Cancer GBDT Accuracy (train):', accuracy(y_pred_train, y_train)*100, '%')
print('Breast Cancer GBDT Accuracy (test):', accuracy(y_pred_test, y_test)*100, '%')
```

```
Breast Cancer GBDT Accuracy (train): 99.74874371859298 %
Breast Cancer GBDT Accuracy (test): 95.32163742690058 %
```