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)

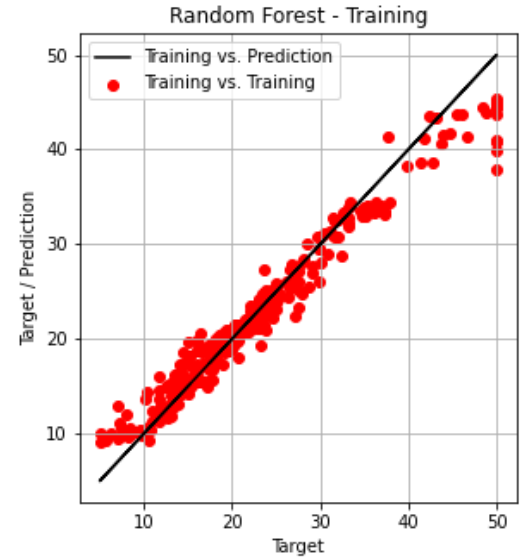
Reference *GBDT.ipynb* for coding implementation of Random Forest code.

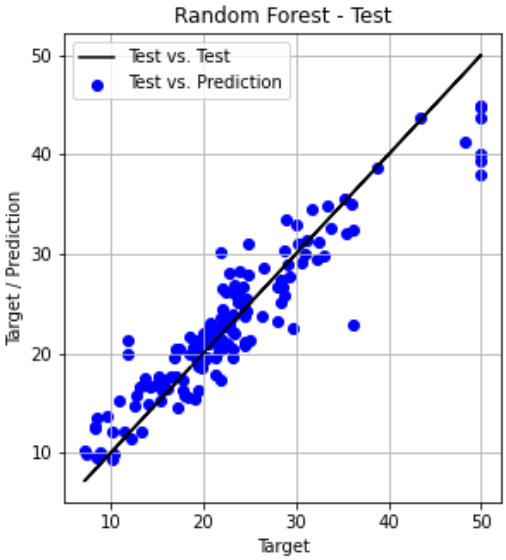
Given the Boston house price dataset, below in the table is the calculated training and test RMSE using the Random Forest object. In addition, the table also included the results from least square regression and ridge regression as calculated in previous homeworks.

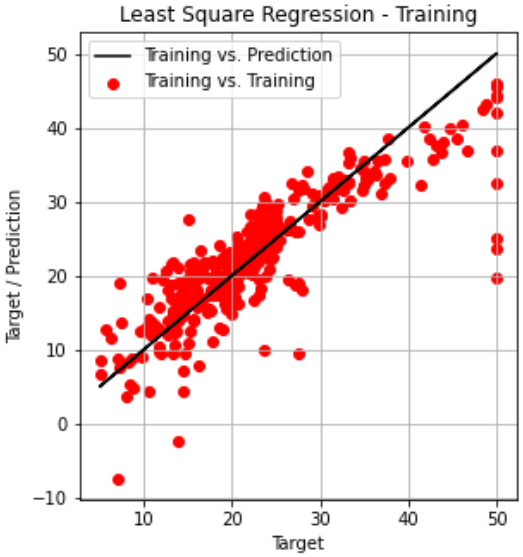
ML Technique	RMSE (Training Data)	RMSE (Test Data)
Random Forest (RF)	2.131	3.343
Least Square Regression	4.8206	5.2092
Ridge Regression	4.8419	5.1474

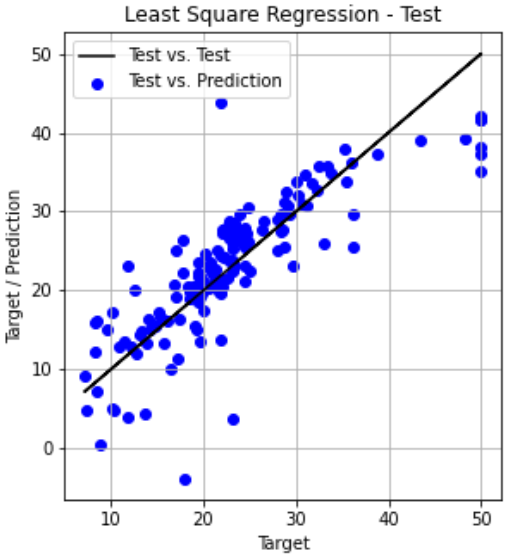
We can see that the Random Forest performs significantly better than both least square and ridge regression based on RMSE score for the training dataset and performs better for the test dataset.

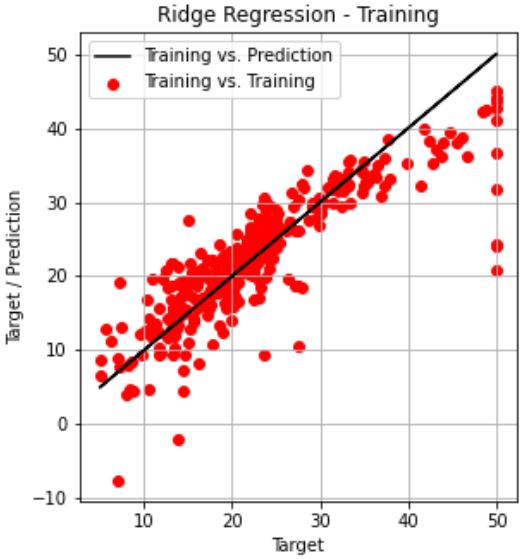
The plots below demonstrate the performance of RF vs. the other regression solutions. The plots below demonstrate how close the predicted value is to the actual value for both our training and test datasets. The line in each graph represents a perfect correlation (i.e. if prediction was equal to actual)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

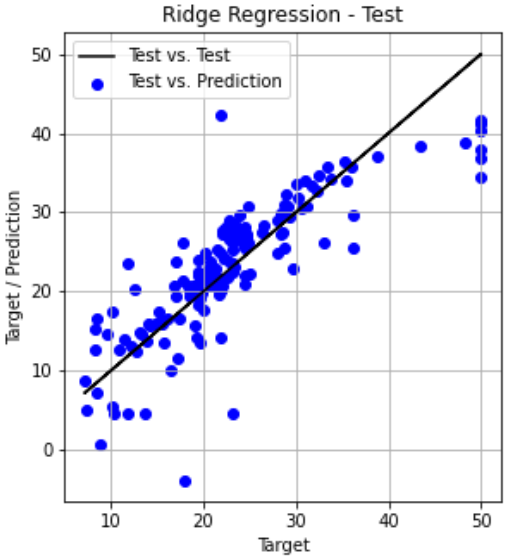

Random Forest - Training
Training vs. Prediction
Training vs. Training
Target / Prediction
Target


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)

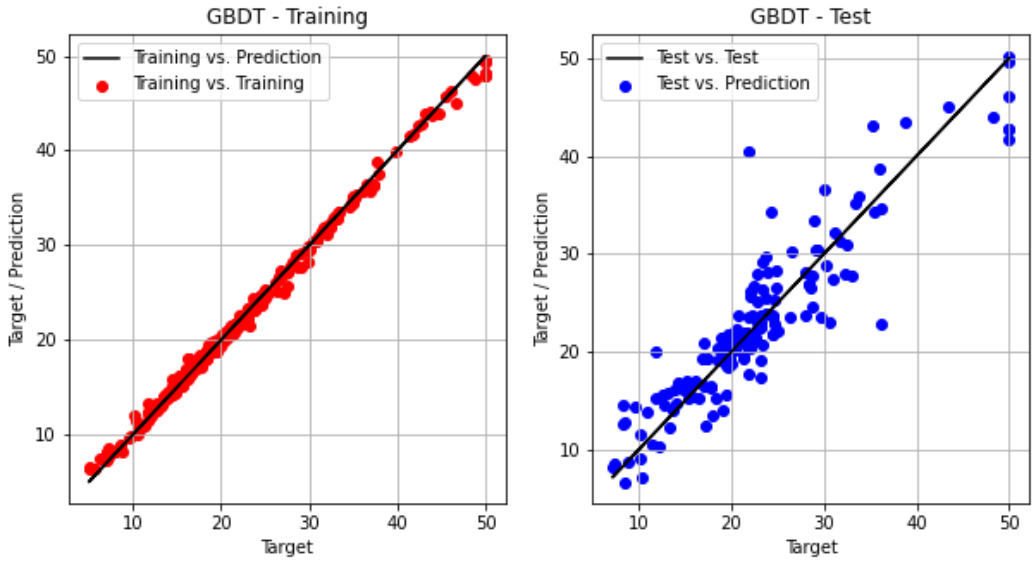
Reference *GBDT.ipynb* for coding implementation of Random Forest code.

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).