ROB313: Assignment 1 Report

Sidhanth Moolayil
1004101679
Feb 9, 2021

1. Objectives

The purpose of this assignment is to introduce regression and classification problems, as well as the kNN and SVD methods to solve them. In this introduction, various norms will also be compared to quantify their respective effectiveness as distance metrics. In these implementations, practical measures such as the KDtree class will also be studied through their impacts on the algorithm's run time compared to the brute force implementation.

2. Code Structure and Employed Strategies

The structure of the code is formatted with several helper functions answering individual question parts, with the main code below where the user can control which questions should be run. Each question is initially set to False, but can be set to True if the user wants to run that particular question.

3. Question 1

Data Table 1

Dataset	Estimated K (1 to 20)	Prefered Distance Function (l1/l2)	Cross-Validation RMS Error	Test RMS Error
mauna_loa	2	L1 norm	0.03491152465200316	0.44070489035463933
rosenbrock	1	L2 norm	0.2749482207897659	0. 289049284976465
pumadyn32nm	17	L1 norm	0.834178497646590	0. 875638033646787

The search strategy for K and the preferred distance function involved a brute force checking of the errors for every K value and distance function and selecting the K and distance function corresponding to the minimum.

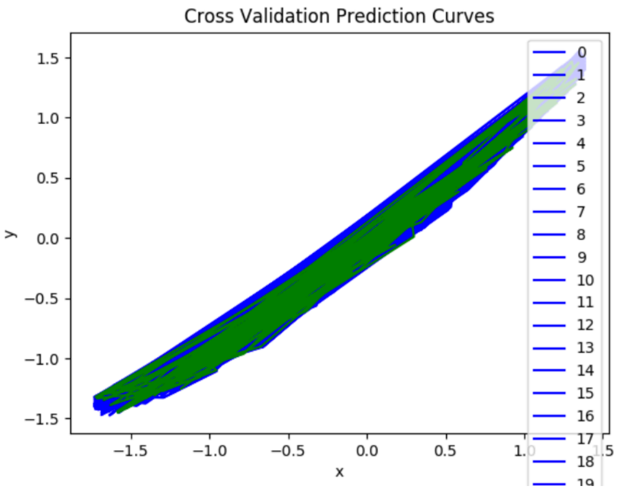


Figure 1. Cross Validation Prediction Cruves

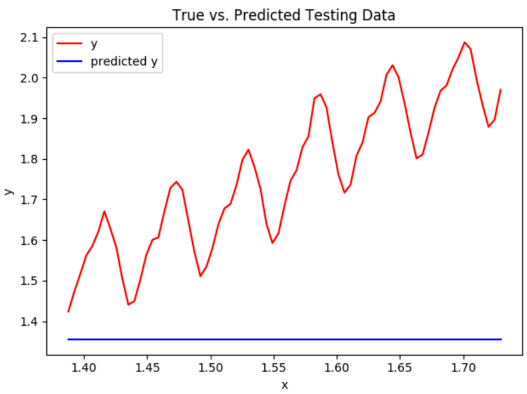


Figure 2. Prediction on Test Set

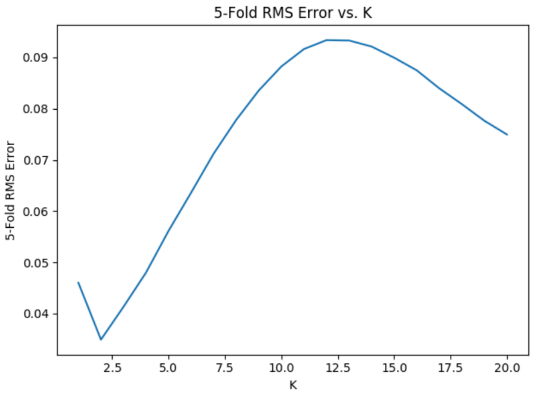


Figure 3. Cross Validation Loss across K values

The figures above for the mauna loa data set shows that while the kNN algorithm is able to reach incredibly low cross validation error, the trained model is ineffective for testing data. The likely reason is the fact that temporal data is used. Comparatively, the rosenbrock dataset had decent performance, with validation errors matching testing errors, while pumadyn32nm, while having a validation error matching its testing, preformed very poorly.

4. Question 2

Data Table 2: Comparison of Accuracies between Brute Force Implementation and KDTree Algorithm of Predicting Testing Data

Dimension	Brute Force Implementation Testing Error	KDTree Algorithm Testing Error
2	0.2826298517051506	0.2826298517051506
3	0.3964060144579797	0.3964060144579797
4	0.42261798365873704	0.42261798365873704
5	0.5541281346915813	0.5541281346915813
6	0.6207218117365524	0.6207218117365524
7	0.6857095048097605	0.6857095048097605

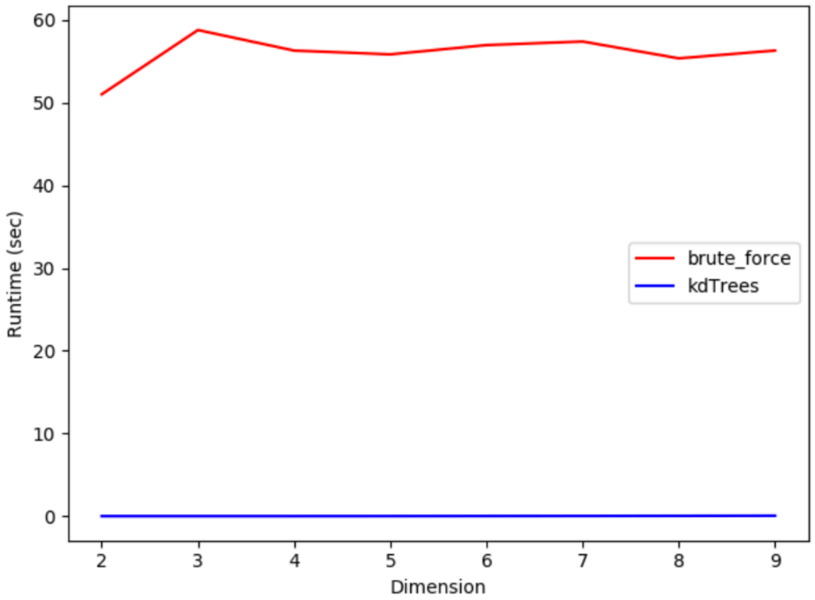


Figure 4. Run-times for Brute Force and KDTree Algorithms across varying dimension on rosenbrock dataset

In terms of accuracy, the brute force implementation preformed similarly to the KDTrees algorithm as expected, however in run time, the KDTrees algorithm was observed to run significantly faster, as reported in Figure 4, with the KDTrees having run times in the order of 10^-3 seconds, and the brute force implementation having run times in the 50-60 second range.

5. Question 3

Data Table 3

Dataset	Estimated K (1 to 20)	Prefered Distance Function (l1/l2)	Validation Accuracy	Test Accuracy
iris	1	L2 norm	0.7741935483870968	1.0
mnist_small	1	L2 norm	0.95	0.958

The search strategy for K and the preferred distance function again involved a brute force checking of the accuracy for every K value and distance function, and selecting the K and distance function corresponding to the minimum.

6. Question 4

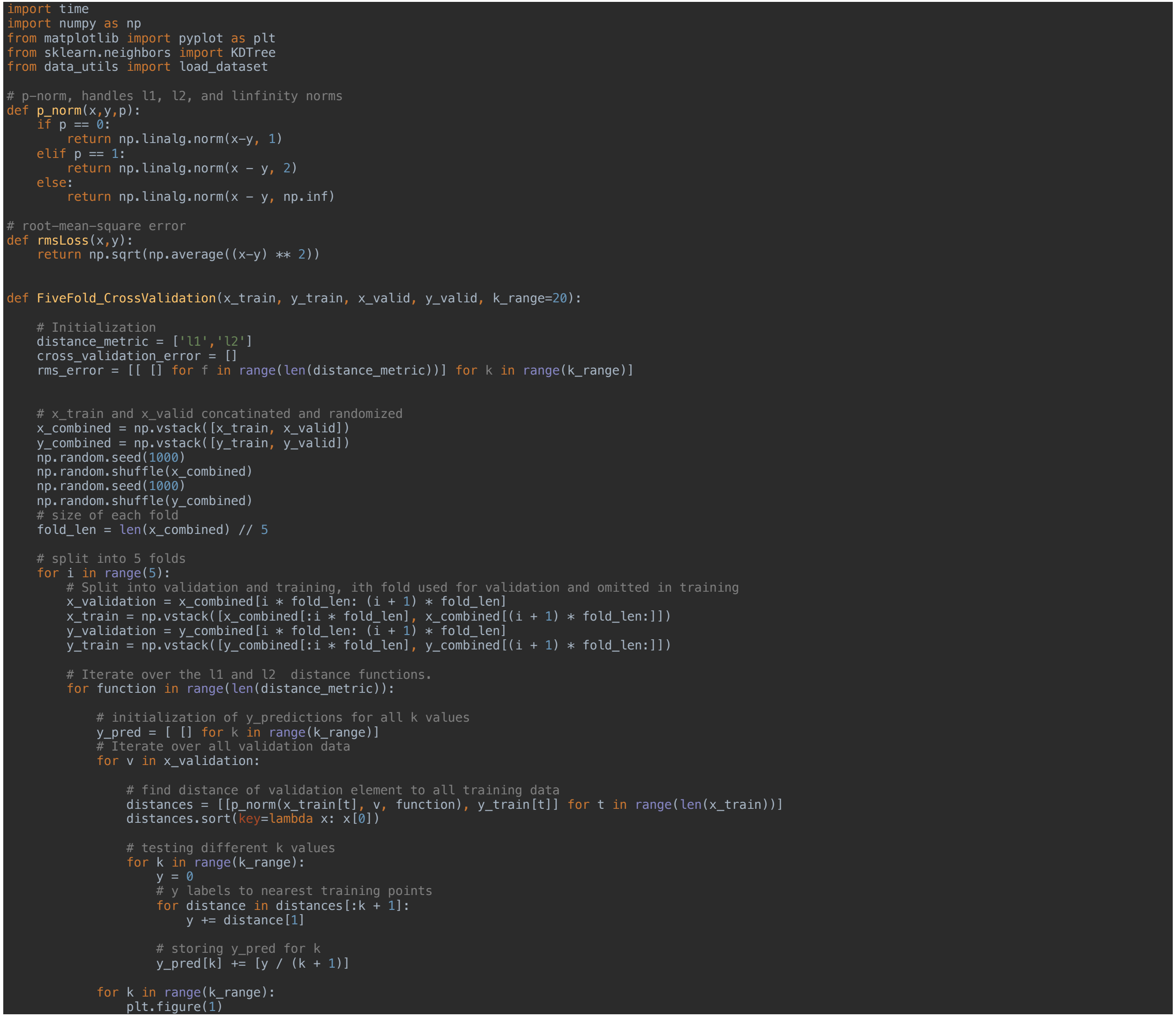
Data Table 4: SVD Testing Results by Dataset

Regression Datasets	**RMS Error**
mauna_loa	0.34938831049910163
rosenbrock	0.9833188519407868
pumadyn32nm	0.8622512436598077
Classification Datasets	**Testing Accuracy**
iris	0.8666666666666667
mnist_small	0.857

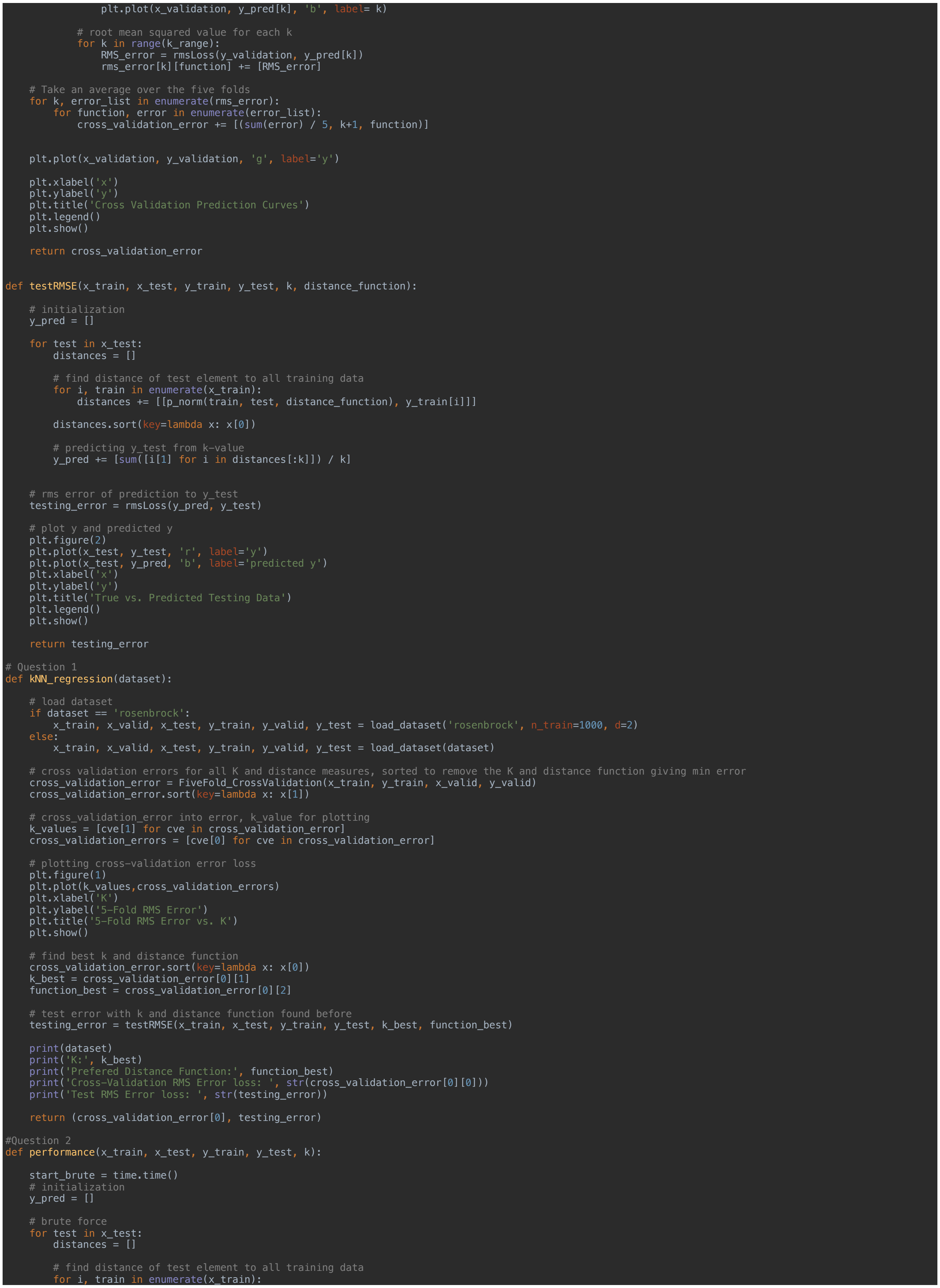
Compared to the accuracies achieved by kNNs (seen in Data Tables 1 and 3), SVD preformed significantly poorer by comparison.

Appendices

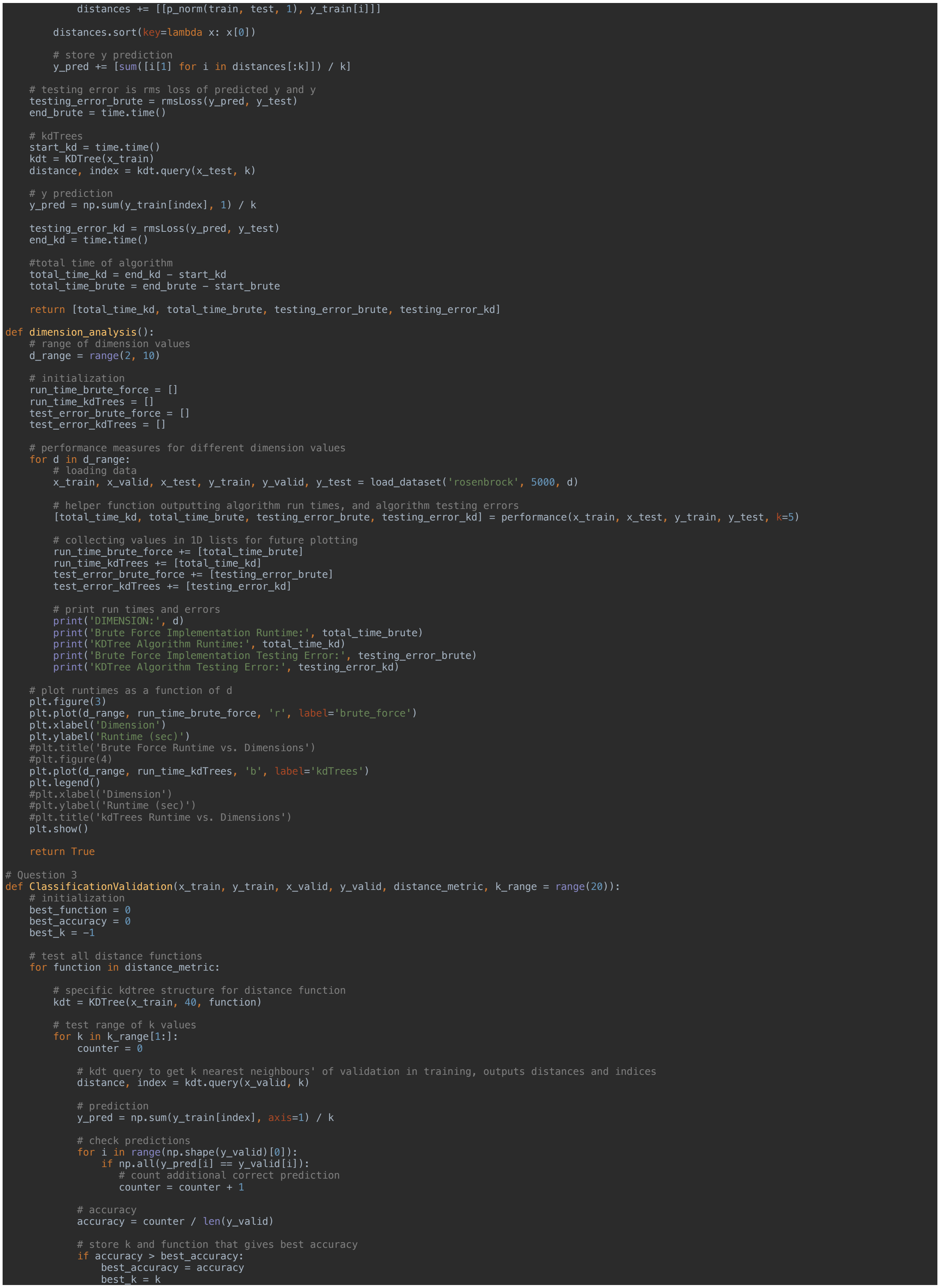
Appendix A: Code



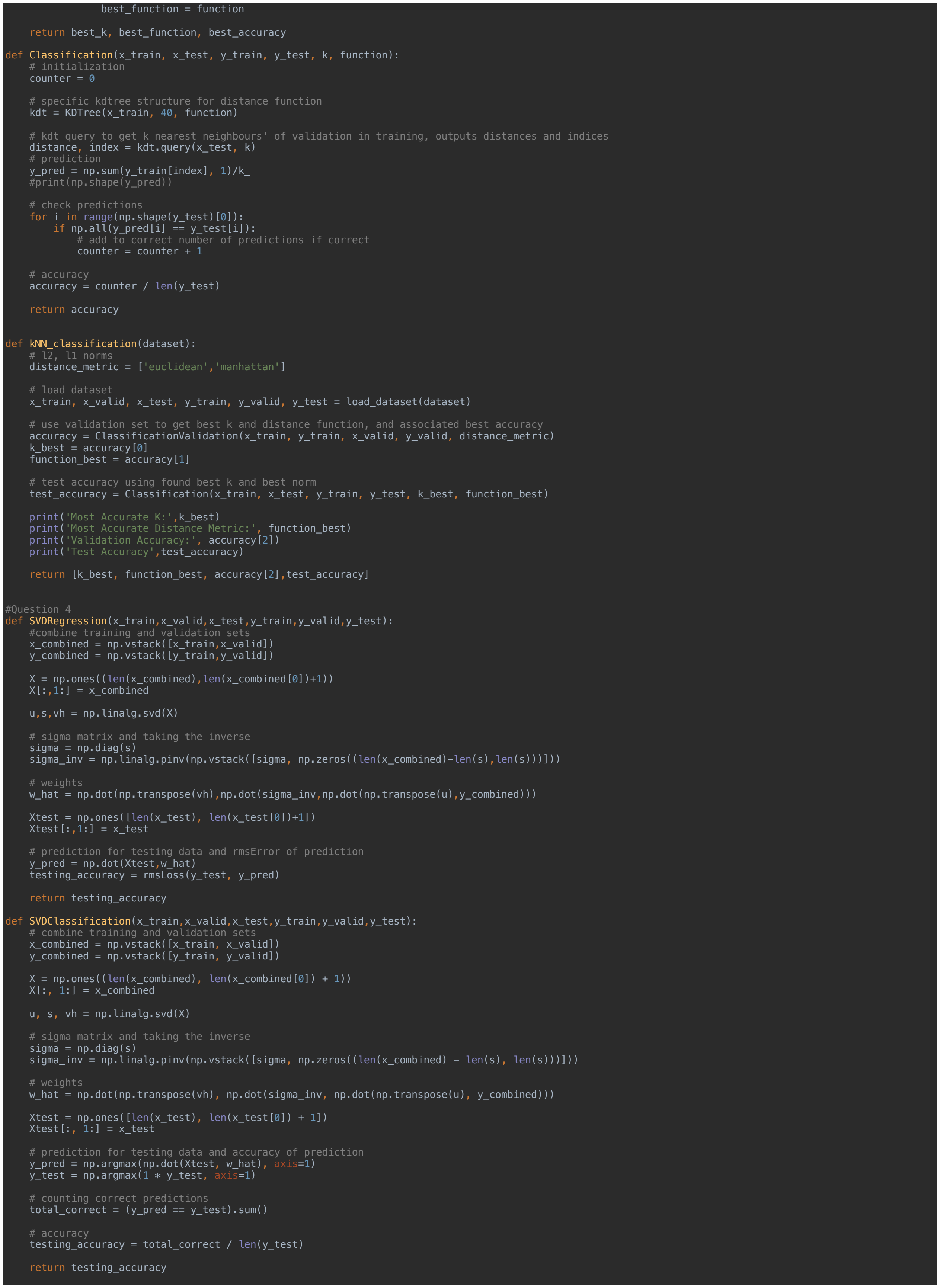
```
import time
import numpy as np
from matplotlib import pyplot as plt
from sklearn.neighbors import KDTree
from data_utils import load_dataset

# p-norm, handles l1, l2, and linfinity norms
def p_norm(x,y,p):
    if p == 0:
        return np.linalg.norm(x-y, 1)
    elif p == 1:
        return np.linalg.norm(x - y, 2)
    else:
        return np.linalg.norm(x - y, np.inf)

# root-mean-square error
def rmsLoss(x,y):
    return np.sqrt(np.average((x-y) ** 2))


def FiveFold_CrossValidation(x_train, y_train, x_valid, y_valid, k_range=20):

    # Initialization
    distance_metric = ['l1','l2']
    cross_validation_error = []
    rms_error = [[ [] for f in range(len(distance_metric))] for k in range(k_range)]


    # x_train and x_valid concatinated and randomized
    x_combined = np.vstack([x_train, x_valid])
    y_combined = np.vstack([y_train, y_valid])
    np.random.seed(1000)
    np.random.shuffle(x_combined)
    np.random.seed(1000)
    np.random.shuffle(y_combined)
    # size of each fold
    fold_len = len(x_combined) // 5

    # split into 5 folds
    for i in range(5):
        # Split into validation and training, ith fold used for validation and omitted in training
        x_validation = x_combined[i * fold_len: (i + 1) * fold_len]
        x_train = np.vstack([x_combined[:i * fold_len], x_combined[(i + 1) * fold_len:]])
        y_validation = y_combined[i * fold_len: (i + 1) * fold_len]
        y_train = np.vstack([y_combined[:i * fold_len], y_combined[(i + 1) * fold_len:]])

        # Iterate over the l1 and l2  distance functions.
        for function in range(len(distance_metric)):

            # initialization of y_predictions for all k values
            y_pred = [ [] for k in range(k_range)]
            # Iterate over all validation data
            for v in x_validation:

                # find distance of validation element to all training data
                distances = [[p_norm(x_train[t], v, function), y_train[t]] for t in range(len(x_train))]
                distances.sort(key=lambda x: x[0])

                # testing different k values
                for k in range(k_range):
                    y = 0
                    # y labels to nearest training points
                    for distance in distances[:k + 1]:
                        y += distance[1]

                    # storing y_pred for k
                    y_pred[k] += [y / (k + 1)]

            for k in range(k_range):
                plt.figure(1)
```



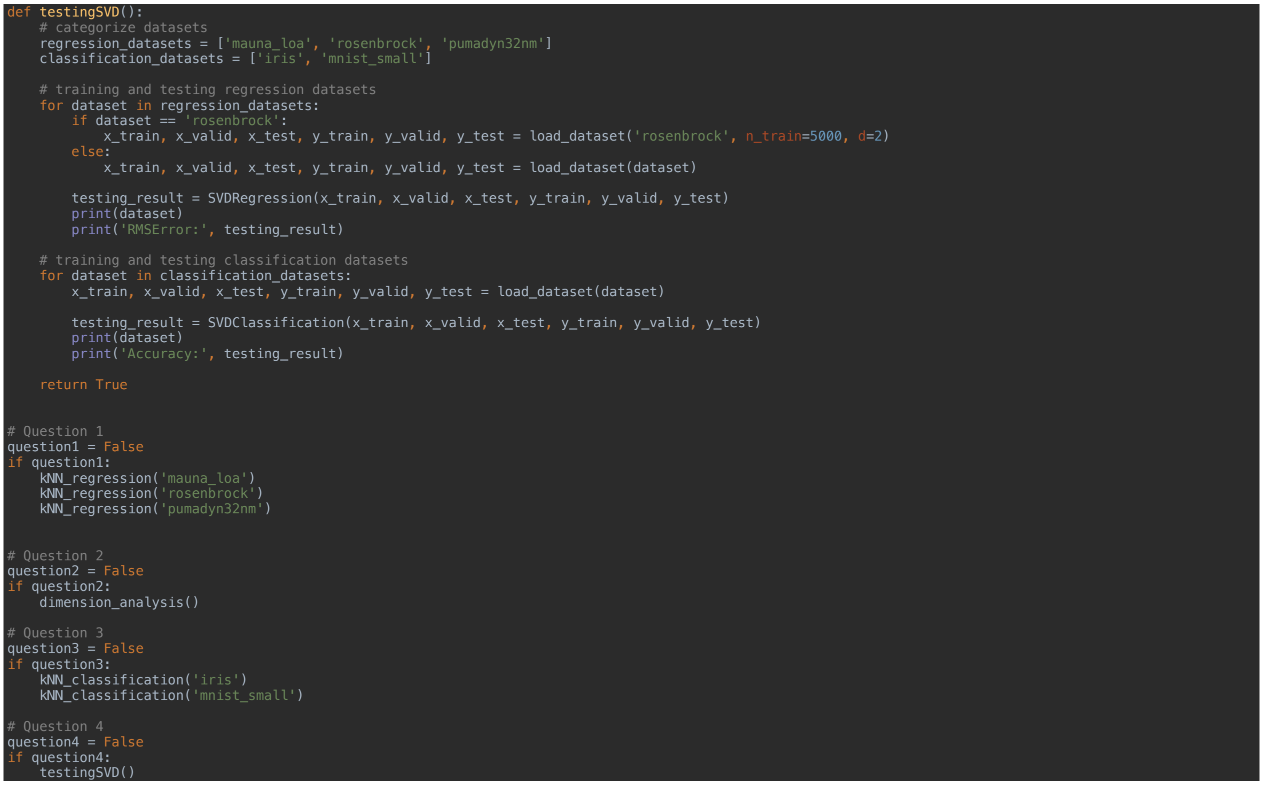
```python
                plt.plot(x_validation, y_pred[k], 'b', label= k)

            # root mean squared value for each k
            for k in range(k_range):
                RMS_error = rmsLoss(y_validation, y_pred[k])
                rms_error[k][function] += [RMS_error]

    # Take an average over the five folds
    for k, error_list in enumerate(rms_error):
        for function, error in enumerate(error_list):
            cross_validation_error += [(sum(error) / 5, k+1, function)]


    plt.plot(x_validation, y_validation, 'g', label='y')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Cross Validation Prediction Curves')
    plt.legend()
    plt.show()

    return cross_validation_error


def testRMSE(x_train, x_test, y_train, y_test, k, distance_function):

    # initialization
    y_pred = []

    for test in x_test:
        distances = []

        # find distance of test element to all training data
        for i, train in enumerate(x_train):
            distances += [[p_norm(train, test, distance_function), y_train[i]]]

        distances.sort(key=lambda x: x[0])

        # predicting y_test from k-value
        y_pred += [sum([i[1] for i in distances[:k]]) / k]


    # rms error of prediction to y_test
    testing_error = rmsLoss(y_pred, y_test)

    # plot y and predicted y
    plt.figure(2)
    plt.plot(x_test, y_test, 'r', label='y')
    plt.plot(x_test, y_pred, 'b', label='predicted y')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('True vs. Predicted Testing Data')
    plt.legend()
    plt.show()

    return testing_error

# Question 1
def kNN_regression(dataset):

    # load dataset
    if dataset == 'rosenbrock':
        x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('rosenbrock', n_train=1000, d=2)
    else:
        x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset)

    # cross validation errors for all K and distance measures, sorted to remove the K and distance function giving min error
    cross_validation_error = FiveFold_CrossValidation(x_train, y_train, x_valid, y_valid)
    cross_validation_error.sort(key=lambda x: x[1])

    # cross_validation_error into error, k_value for plotting
    k_values = [cve[1] for cve in cross_validation_error]
    cross_validation_errors = [cve[0] for cve in cross_validation_error]

    # plotting cross-validation error loss
    plt.figure(1)
    plt.plot(k_values,cross_validation_errors)
    plt.xlabel('K')
    plt.ylabel('5-Fold RMS Error')
    plt.title('5-Fold RMS Error vs. K')
    plt.show()

    # find best k and distance function
    cross_validation_error.sort(key=lambda x: x[0])
    k_best = cross_validation_error[0][1]
    function_best = cross_validation_error[0][2]

    # test error with k and distance function found before
    testing_error = testRMSE(x_train, x_test, y_train, y_test, k_best, function_best)

    print(dataset)
    print('K:', k_best)
    print('Prefered Distance Function:', function_best)
    print('Cross-Validation RMS Error loss: ', str(cross_validation_error[0][0]))
    print('Test RMS Error loss: ', str(testing_error))

    return (cross_validation_error[0], testing_error)

#Question 2
def performance(x_train, x_test, y_train, y_test, k):

    start_brute = time.time()
    # initialization
    y_pred = []

    # brute force
    for test in x_test:
        distances = []

        # find distance of test element to all training data
        for i, train in enumerate(x_train):
```



```
            distances += [[p_norm(train, test, 1), y_train[i]]]

        distances.sort(key=lambda x: x[0])

        # store y prediction
        y_pred += [sum([i[1] for i in distances[:k]]) / k]

    # testing error is rms loss of predicted y and y
    testing_error_brute = rmsLoss(y_pred, y_test)
    end_brute = time.time()

    # kdTrees
    start_kd = time.time()
    kdt = KDTree(x_train)
    distance, index = kdt.query(x_test, k)

    # y prediction
    y_pred = np.sum(y_train[index], 1) / k

    testing_error_kd = rmsLoss(y_pred, y_test)
    end_kd = time.time()

    #total time of algorithm
    total_time_kd = end_kd - start_kd
    total_time_brute = end_brute - start_brute

    return [total_time_kd, total_time_brute, testing_error_brute, testing_error_kd]

def dimension_analysis():
    # range of dimension values
    d_range = range(2, 10)

    # initialization
    run_time_brute_force = []
    run_time_kdTrees = []
    test_error_brute_force = []
    test_error_kdTrees = []

    # performance measures for different dimension values
    for d in d_range:
        # loading data
        x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('rosenbrock', 5000, d)

        # helper function outputting algorithm run times, and algorithm testing errors
        [total_time_kd, total_time_brute, testing_error_brute, testing_error_kd] = performance(x_train, x_test, y_train, y_test, k=5)

        # collecting values in 1D lists for future plotting
        run_time_brute_force += [total_time_brute]
        run_time_kdTrees += [total_time_kd]
        test_error_brute_force += [testing_error_brute]
        test_error_kdTrees += [testing_error_kd]

        # print run times and errors
        print('DIMENSION:', d)
        print('Brute Force Implementation Runtime:', total_time_brute)
        print('KDTree Algorithm Runtime:', total_time_kd)
        print('Brute Force Implementation Testing Error:', testing_error_brute)
        print('KDTree Algorithm Testing Error:', testing_error_kd)

    # plot runtimes as a function of d
    plt.figure(3)
    plt.plot(d_range, run_time_brute_force, 'r', label='brute_force')
    plt.xlabel('Dimension')
    plt.ylabel('Runtime (sec)')
    #plt.title('Brute Force Runtime vs. Dimensions')
    #plt.figure(4)
    plt.plot(d_range, run_time_kdTrees, 'b', label='kdTrees')
    plt.legend()
    #plt.xlabel('Dimension')
    #plt.ylabel('Runtime (sec)')
    #plt.title('kdTrees Runtime vs. Dimensions')
    plt.show()

    return True

# Question 3
def ClassificationValidation(x_train, y_train, x_valid, y_valid, distance_metric, k_range = range(20)):
    # initialization
    best_function = 0
    best_accuracy = 0
    best_k = -1

    # test all distance functions
    for function in distance_metric:

        # specific kdtree structure for distance function
        kdt = KDTree(x_train, 40, function)

        # test range of k values
        for k in k_range[1:]:
            counter = 0

            # kdt query to get k nearest neighbours' of validation in training, outputs distances and indices
            distance, index = kdt.query(x_valid, k)

            # prediction
            y_pred = np.sum(y_train[index], axis=1) / k

            # check predictions
            for i in range(np.shape(y_valid)[0]):
                if np.all(y_pred[i] == y_valid[i]):
                    # count additional correct prediction
                    counter = counter + 1

            # accuracy
            accuracy = counter / len(y_valid)

            # store k and function that gives best accuracy
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_k = k
```



```python
                best_function = function

    return best_k, best_function, best_accuracy

def Classification(x_train, x_test, y_train, y_test, k, function):
    # initialization
    counter = 0

    # specific kdtree structure for distance function
    kdt = KDTree(x_train, 40, function)

    # kdt query to get k nearest neighbours' of validation in training, outputs distances and indices
    distance, index = kdt.query(x_test, k)
    # prediction
    y_pred = np.sum(y_train[index], 1)/k_
    #print(np.shape(y_pred))

    # check predictions
    for i in range(np.shape(y_test)[0]):
        if np.all(y_pred[i] == y_test[i]):
            # add to correct number of predictions if correct
            counter = counter + 1

    # accuracy
    accuracy = counter / len(y_test)

    return accuracy


def kNN_classification(dataset):
    # l2, l1 norms
    distance_metric = ['euclidean','manhattan']

    # load dataset
    x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset)

    # use validation set to get best k and distance function, and associated best accuracy
    accuracy = ClassificationValidation(x_train, y_train, x_valid, y_valid, distance_metric)
    k_best = accuracy[0]
    function_best = accuracy[1]

    # test accuracy using found best k and best norm
    test_accuracy = Classification(x_train, x_test, y_train, y_test, k_best, function_best)

    print('Most Accurate K:',k_best)
    print('Most Accurate Distance Metric:', function_best)
    print('Validation Accuracy:', accuracy[2])
    print('Test Accuracy',test_accuracy)

    return [k_best, function_best, accuracy[2],test_accuracy]


#Question 4
def SVDRegression(x_train,x_valid,x_test,y_train,y_valid,y_test):
    #combine training and validation sets
    x_combined = np.vstack([x_train,x_valid])
    y_combined = np.vstack([y_train,y_valid])

    X = np.ones((len(x_combined),len(x_combined[0])+1))
    X[:,1:] = x_combined

    u,s,vh = np.linalg.svd(X)

    # sigma matrix and taking the inverse
    sigma = np.diag(s)
    sigma_inv = np.linalg.pinv(np.vstack([sigma, np.zeros((len(x_combined)-len(s),len(s)))]))

    # weights
    w_hat = np.dot(np.transpose(vh),np.dot(sigma_inv,np.dot(np.transpose(u),y_combined)))

    Xtest = np.ones([len(x_test), len(x_test[0])+1])
    Xtest[:,1:] = x_test

    # prediction for testing data and rmsError of prediction
    y_pred = np.dot(Xtest,w_hat)
    testing_accuracy = rmsLoss(y_test, y_pred)

    return testing_accuracy

def SVDClassification(x_train,x_valid,x_test,y_train,y_valid,y_test):
    # combine training and validation sets
    x_combined = np.vstack([x_train, x_valid])
    y_combined = np.vstack([y_train, y_valid])

    X = np.ones((len(x_combined), len(x_combined[0]) + 1))
    X[:, 1:] = x_combined

    u, s, vh = np.linalg.svd(X)

    # sigma matrix and taking the inverse
    sigma = np.diag(s)
    sigma_inv = np.linalg.pinv(np.vstack([sigma, np.zeros((len(x_combined) - len(s), len(s)))]))

    # weights
    w_hat = np.dot(np.transpose(vh), np.dot(sigma_inv, np.dot(np.transpose(u), y_combined)))

    Xtest = np.ones([len(x_test), len(x_test[0]) + 1])
    Xtest[:, 1:] = x_test

    # prediction for testing data and accuracy of prediction
    y_pred = np.argmax(np.dot(Xtest, w_hat), axis=1)
    y_test = np.argmax(1 * y_test, axis=1)

    # counting correct predictions
    total_correct = (y_pred == y_test).sum()

    # accuracy
    testing_accuracy = total_correct / len(y_test)

    return testing_accuracy
```



```python
def testingSVD():
    # categorize datasets
    regression_datasets = ['mauna_loa', 'rosenbrock', 'pumadyn32nm']
    classification_datasets = ['iris', 'mnist_small']

    # training and testing regression datasets
    for dataset in regression_datasets:
        if dataset == 'rosenbrock':
            x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('rosenbrock', n_train=5000, d=2)
        else:
            x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset)

        testing_result = SVDRegression(x_train, x_valid, x_test, y_train, y_valid, y_test)
        print(dataset)
        print('RMSError:', testing_result)

    # training and testing classification datasets
    for dataset in classification_datasets:
        x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset(dataset)

        testing_result = SVDClassification(x_train, x_valid, x_test, y_train, y_valid, y_test)
        print(dataset)
        print('Accuracy:', testing_result)

    return True


# Question 1
question1 = False
if question1:
    kNN_regression('mauna_loa')
    kNN_regression('rosenbrock')
    kNN_regression('pumadyn32nm')


# Question 2
question2 = False
if question2:
    dimension_analysis()

# Question 3
question3 = False
if question3:
    kNN_classification('iris')
    kNN_classification('mnist_small')

# Question 4
question4 = False
if question4:
    testingSVD()
```

Appendix B: Raw Code Output

```
Question 1
mauna_loa
K: 2
Prefered Distance Function: 0
Cross-Validation RMS Error loss:  0.03491152465200316
Test RMS Error loss:  0.44070489035463933

rosenbrock
K: 1
Prefered Distance Function: 1
Cross-Validation RMS Error:  0.2749482207897659
Test RMS Error:  0.289049284976465

pumadyn32nm
K: 17
Prefered Distance Function: 0
Cross-Validation RMS Error:  0.834178497646590
Test RMS Error:  0.875638033646787

Question 2
DIMENSION: 2
Brute Force Implementation Runtime: 45.54945111274719
KDTree Algorithm Runtime: 0.007797956466674805
Brute Force Implementation Testing Error: 0.2826298517051506
KDTree Algorithm Testing Error: 0.2826298517051506

DIMENSION: 3
Brute Force Implementation Runtime: 46.16702914237976
KDTree Algorithm Runtime: 0.005339145660400391
Brute Force Implementation Testing Error: 0.3964060144579797
KDTree Algorithm Testing Error: 0.3964060144579797
```

```
DIMENSION: 4
Brute Force Implementation Runtime: 46.4911003112793
KDTree Algorithm Runtime: 0.008162736892700195
Brute Force Implementation Testing Error: 0.42261798365873704
KDTree Algorithm Testing Error: 0.42261798365873704

DIMENSION: 5
Brute Force Implementation Runtime: 47.658610343933105
KDTree Algorithm Runtime: 0.012826919555664062
Brute Force Implementation Testing Error: 0.5541281346915813
KDTree Algorithm Testing Error: 0.5541281346915813

DIMENSION: 6
Brute Force Implementation Runtime: 46.12748122215271
KDTree Algorithm Runtime: 0.020218849182128906
Brute Force Implementation Testing Error: 0.6207218117365524
KDTree Algorithm Testing Error: 0.6207218117365524

DIMENSION: 7
Brute Force Implementation Runtime: 54.50399875640869
KDTree Algorithm Runtime: 0.037404775619506836
Brute Force Implementation Testing Error: 0.6857095048097605
KDTree Algorithm Testing Error: 0.6857095048097605

DIMENSION: 8
Brute Force Implementation Runtime: 49.57274389266968
KDTree Algorithm Runtime: 0.09354686737060547
Brute Force Implementation Testing Error: 0.7553856555694674
KDTree Algorithm Testing Error: 0.7553856555694674

DIMENSION: 9
Brute Force Implementation Runtime: 50.68706011772156
KDTree Algorithm Runtime: 0.07958507537841797
Brute Force Implementation Testing Error: 0.8017483490670231
KDTree Algorithm Testing Error: 0.8017483490670231
```

Question 3

```
iris
Most Accurate K: 1
Most Accurate Distance Metric: euclidean
Validation Accuracy: 0.7741935483870968
Test Accuracy 1.0

mnist_small
Most Accurate K: 1
Most Accurate Distance Metric: euclidean
Validation Accuracy: 0.95
Test Accuracy 0.958
```

Question 4

```
mauna_loa
RMSError: 0.34938831049910163
rosenbrock
RMSError: 0.9833188519407868
pumadyn32nm
RMSError: 0.8622512436598077
iris
Accuracy: 0.8666666666666667
mnist_small
Accuracy: 0.857
```